PROYECTO FIN DE CARRERA DE INGENIERÍA INFORMÁTICA

Sistemas Informáticos, Facultad de Informática,
Universidad Complutense de Madrid

Mejora de la Evaluación de Expresiones Regulares Sobre Hardware Reconfigurable

---

Autor: Claudio Alejandro Muñoz Fernández

Directores: Guadalupe Miñana Ropero y Marcos Sánchez-Élez Martín

Curso académico 2010-2011

AUTORIZACIÓN DE DIFUSIÓN

El abajo firmante, matriculado en la asignatura Sistemas Informáticos de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Proyecto Fin de Carrera: "Evaluación de Expresiones Regulares sobre Hardware Reconfigurable", realizado durante el curso académico 2010-2011 bajo la dirección de Guadalupe Miñana Ropero y Marcos Sánchez-Élez Martín en el Departamento de Arquitectura de Computadores y Automática.

CONTENTS

ABSTRACT

Since the Internet was born, the amount of data that systems process has increased in an exponential way and this is the reason because these systems need to be fast, flexible and powerful. Nowadays, communications keep increasing the speed requirements for data processing, and the FPGA's are ideal for this task.

In data processing, a huge amount of time is dedicated to pattern matching, frequently involving regular expressions matching. As the amount of patterns to be checked grow up, so does the hardware complexity dedicated to its recognition. Thus it needs to be flexible to be able to adapt to the necessary changes with ease.

In this project a VHDL code generator implemented in Java is presented. The code generated describes a regular expressions recognizer of various sets given by parameter, which will be synthetized by an FPGA. This module takes various sets of regular expressions and generates the VHDL code that describes the system which recognizes them.

The code generator is flexible, due to great modularity and upgradeability that software offers. Thus, the main advantage of this model consists on the possibility of combining the flexibility of software with the speed of hardware in order to create fast and low cost recognizers in a flexible and easy way.

KEY WORD LIST

Regular expression matching, pattern matching, code generator, regular expression optimization, Reconfigurable Hardware, deep packet inspection, Networking, VHDL.

Introducción y objetivos

En informática, la búsqueda de patrones es el acto de comprobación de una secuencia de tokens con el fin de verificar si cumplen determinadas características.

Hoy en día, la principal forma de tratar con expresiones regulares es utilizando soluciones software. Esto es muy flexible, pero conlleva una falta de rendimiento, y con la cantidad de patrones a reconocer aumentando cada día, necesitamos soluciones rápidas, simples y de bajo coste para su reconocimiento.

En este proyecto se presenta una solución para el reconocimiento de expresiones regulares que combina la flexibilidad del software con la velocidad del hardware. El hecho de combinar los conocimientos adquiridos en diferentes áreas a lo largo de la carrera hace este proyecto algo diferente del resto.

Por un lado tenemos la necesidad de tratar con expresiones regulares, estudiadas en Teoría de Autómatas y Lenguajes Formales, combinada con un buen conocimiento de estructuras de datos, metodología y tecnología de programación y programación en Java. Por otro lado tenemos que saber cómo generar código VHDL sintetizable, por lo que es necesario un buen conocimiento de Arquitectura de Computadores, diseño lógico y diseño de hardware reconfigurable usando FPGAs.

El objetivo principal de este proyecto es minimizar la cantidad de lógica generada para reconocer expresiones regulares, lo que nos permite maximizar el número de éstas procesadas por la FPGA.

Como se muestra en la figura 1, el generador de código, que está escrito en Java, acepta un archivo de texto con varios conjuntos de expresiones regulares y posteriormente los procesa, generando el archivo VHDL que describe al reconocedor.

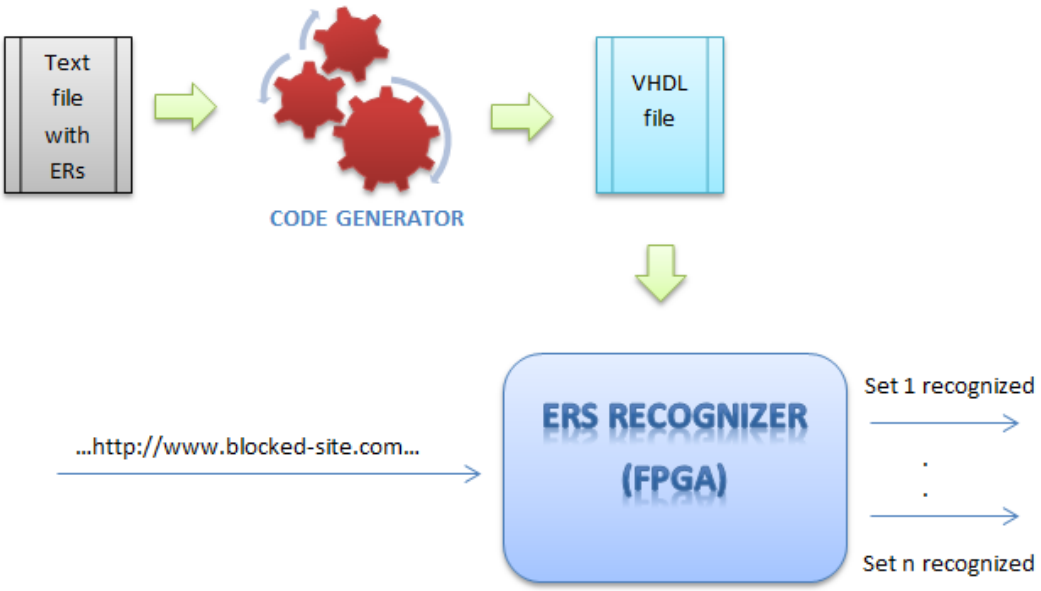

Figura 1 - Diagrama de la arquitectura

El reconocedor propuesto en este proyecto tiene n salidas, una por cada conjunto. Cuando se detecta que la cadena de entrada coincide con algún patrón, se activa la salida correspondiente. De esta forma, se puede detectar el conjunto al que ésta pertenece.

Llamaremos engine a cada módulo encargado de reconocer una expresión regular dada (para más información consultar el trabajo previo realizado por Ignacio en la página 30).

El generador de código se encarga principalmente del proceso de generación de hardware, minimizando la cantidad de lógica usada. La minimización de hardware se ha logrado en dos pasos: en primer lugar, reutilización completa de engines, y en segundo lugar, reutilización parcial.

La reutilización completa de engines consiste en volver a usar el mismo engine cuando una expresión regular se utiliza en varios conjuntos. Esto conduce a importantes optimizaciones, evitando la necesidad de generar módulos duplicados.

La reutilización parcial se basa en la idea misma utilizada para la reutilización completa, pero sin limitarnos a una expresión regular entera. Si una expresión regular tiene una parte común con cualquier otra existente en cualquier otro conjunto, el hardware necesario para esa parte no se genera, utilizando en su lugar la lógica existente (nótese que esto tiene algunas limitaciones que se explicarán más adelante en la página 60). De esta manera, compartimos el hardware de cada engine con cualquier otro, generando sólo la nueva lógica necesaria. Esto aumenta la cantidad de hardware ahorrado, dada la gran cantidad de optimizaciones realizadas cuando se dan las condiciones adecuadas (ver página 54).

Para hacer frente a los objetivos propuestos para este proyecto, se ha realizado un análisis exhaustivo del trabajo previo. Esto ha implicado el estudio del trabajo realizado por Sidhu y Prasanna [15] (ver página 27), así como la implementación de Ignacio basada en engines [14]. Además, ha habido una revisión previa del estado del arte para asegurar que ésta es una buena solución en comparación con otras existentes en la actualidad.

Sidhu y Prasanna [15] presentan un método eficiente para el reconocimiento rápido de expresiones regulares utilizando FPGAs, encontrado todas las cadenas de caracteres de un texto dado que reconoce una expresión regular dada.

El proceso de investigación y desarrollo se ha dividido en varias fases, lo que garantiza que todos los objetivos propuestos para cada plazo se han cumplido, de forma similar al proceso de desarrollo de software Scrum [18] [19] utilizado en Ingeniería de Software.

En cada fase se han considerado varias alternativas de diseño, evaluando su simplicidad, coste e implicaciones en otros módulos y futuras mejoras. Esto hace que el análisis sea una parte muy importante del proceso de desarrollo, y aunque conlleva una cantidad importante de tiempo, también previene errores futuros y simplifica la fase de implementación.

Hardware reconfigurable

Una FPGA es un circuito integrado que puede ser programado (con código HDL) para convertirse en casi cualquier tipo de circuito digital. Las FPGAs constan de matrices de bloques lógicos programables de diferentes tipos rodeados de lógica de enrutamiento, lo que permite a los bloques ser programablemente interconectados. La matriz de bloques está rodeada por bloques de entrada / salida que también pueden ser programados.

La arquitectura de las celdas lógicas varía entre las distintas familias de dispositivos. En términos generales, cada celda lógica combina unas pocas entradas binarias (normalmente entre 3 y 10) con una o dos salidas de acuerdo a una función lógica booleana especificada en el programa del usuario. En la mayoría de las familias, el usuario también tiene la opción de fijar la salida combinacional de la celda, de modo que la lógica secuencial también puede ser fácilmente implementada.

La lógica combinacional de las celdas puede ser implementada físicamente como una tabla de consulta (LUT) o como un conjunto de multiplexores y puertas. Los dispositivos LUT son un poco más flexibles y ofrecen más entradas por celda que los multiplexores a expensas de un mayor retardo de propagación.

La función de la FPGA está definida mediante un programa de definido por el usuario. Dependiendo del dispositivo, el programa es grabado de forma permanente o semi-permanente, como parte de un proceso de ensamblaje de la placa, o se bien carga desde una memoria externa cada vez que se enciende el equipo. Esta programabilidad ofrece al usuario acceso a complejos diseños integrados, sin los altos costes de ingeniería asociados con el uso de circuitos integrados específicos.

Definir individualmente todas las conexiones de una FPGA sería una tarea de enormes proporciones. Afortunadamente, se lleva a cabo mediante un software especial. El software traduce los diagramas esquemáticos del usuario o el código del lenguaje de descripción hardware y a continuación coloca y conecta

el diseño traducido. La mayoría estos programas tienen la opción de permitir al usuario configurar la aplicación, colocación y de enrutamiento para obtener un mejor rendimiento y utilización del dispositivo.

Las bibliotecas de macros con funciones más complejas (por ejemplo, sumadores, multiplicadores, multiplexores, etc) simplifican aún más el proceso de diseño, ofreciendo los circuitos más habituales ya optimizados para velocidad o área.

Sin embargo, no todo HDL es sintetizable eficientemente, ya que depende de las herramientas asociadas a la plataforma.

Entorno de desarrollo

En este capítulo se muestra el entorno de desarrollo utilizado, destacando las diferentes herramientas usadas en los procesos de desarrollo y pruebas.

Como ya se mencionó anteriormente, este proyecto combina la flexibilidad del software con la velocidad del hardware, por lo que debemos elegir un entorno de desarrollo potente y fácil de usar.

En nuestro caso, debido a la complejidad del proyecto, los procesos de desarrollo, depuración y análisis se han dividido en varias partes, con especial énfasis en la modularidad de cada uno.

Como puede verse en la figura 1, el generador de código (escrito en Java) acepta un archivo de texto con conjuntos de expresiones regulares, procesándolos a continuación, y generando el archivo VHDL que describe el reconocedor.

El fichero de texto de entrada se estructura en líneas (conjuntos de expresiones regulares), comenzando cada una por el nombre del conjunto, y seguida por todas sus expresiones regulares separadas por punto y coma.

Para programar el generador de código VHDL se ha utilizado Eclipse [50]. Por otro lado, para depurar y probar el código VHDL generado se ha usado Xilinx ISE [51], además de Xilinx ISIM y Modelsim para simular el diseño.

Eclipse es un entorno de programación software multi-lenguaje que incluye un entorno de desarrollo integrado (IDE) y un sistema de plug-ins extensible. Se puede utilizar para desarrollar aplicaciones en Java y otros lenguajes de programación.

El archivo VHDL se puede utilizar con cualquier herramienta de software para síntesis, análisis y simulación de diseños VHDL. Esto permite:

- Comprobar si hay errores de sintaxis en el código VHDL.
- Ver el esquema del diseño.
- Simular el diseño para una placa específica (por ejemplo, Virtex 5, Spartan 3, etc.)
- Obtener el informe del reloj (por ejemplo, tiempo de ciclo mínimo, máximo retardo combinacional, etc.)
- Calcular la cantidad de lógica generada.
- Ver la cantidad de recursos utilizados de la FPGA.
- Generar el archivo de programación utilizado para configurar el dispositivo de destino.

Xilinx ISE es una herramienta de diseño para la síntesis y el análisis de código HDL que permite elegir y configurar la placa de destino con el fin de simular y sintetizar el diseño correctamente. Esto es muy útil ya que permite realizar ambas tareas con facilidad, así como generar el archivo de programación de la FPGA.

Añadiendo soporte para el reconocimiento de múltiples conjuntos

La primera modificación hecha sobre el trabajo de Ignacio es la adición de soporte para el reconocimiento de múltiples conjuntos.

Como se muestra en la figura 1, el reconocedor tiene n salidas, una por cada conjunto. Cuando una cadena es detectada, la salida correspondiente lo notifica. La figura 2 muestra el esquema para un reconocedor de dos salidas.

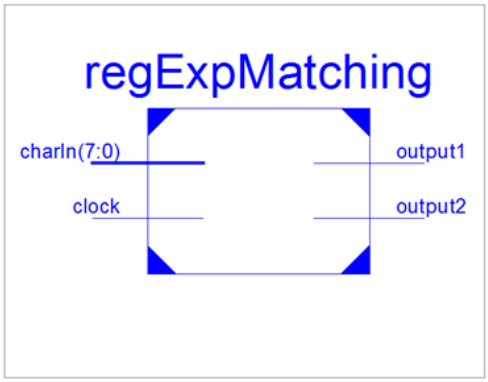

Figura 2 - Módulo del nivel superior

En esta primera modificación, el reconocedor está estructurado en varias partes, cada una dedicada a reconocer un conjunto, y no comparten lógica alguna. Por tanto, ahora existen varias salidas, tantas como conjuntos, siendo así capaces de detectar a la pertenencia a un conjunto determinado de cada cadena de entrada.

Una vez realizada esta mejora, se lleva a cabo el proceso de optimización de hardware, compuesto de dos partes: la reutilización parcial y la reutilización completa de engines (ver páginas 42 y 44).

En la siguiente sección se explica la primera parte del proceso de optimización de hardware.

Reutilización completa de engines

Una vez añadido soporte para el reconocimiento de varios conjuntos, deseamos reutilizar toda la lógica generada a reconocer una expresión regular de un conjunto en otro en lugar de replicarlo. Ésta es la primera mejora implementada con el fin de reutilizar hardware.

La reutilización completa de engines consiste en usar uno ya existente cuando una expresión regular de un conjunto dado está contenida en otro, evitando así tener que volver a generar la misma lógica. Esto da lugar a grandes mejoras en el área utilizada cuando la misma expresión regular está presente en varios conjuntos.

Este proceso se logra comparando todas las formas postfijas de las expresiones regulares de cada conjunto con las demás. Si alguna se ha generado previamente, la salida del engine existente es conectada a las salidas del conjunto actual, sin necesidad de duplicar engines.

Reutilización parcial de engines

Esta fase es bastante más complicada de implementar que el resto, ya que existen algunas limitaciones en el algoritmo de Sidhu y Prasanna [15] y la implementación Ignacio [14] (ver restricciones en la página 60), que afectan al tipo de optimizaciones que se pueden hacer.

La mayoría de estas limitaciones se han resuelto, como se explica a lo largo de este capítulo, lo cual nos permite ahorrar una importante cantidad hardware (ver resultados experimentales en la página 61).

Como se ha comentado antes, la reutilización parcial de engines se basa en la misma idea utilizada para la reutilización completa, pero sin limitarnos a una expresión regular completa. Cuando una expresión tiene una parte común con cualquier otra existente de cualquier conjunto, no se genera el hardware necesario para reconocer esa parte, usando en su lugar la lógica previamente

generada. Por lo tanto, podemos compartir hardware entre todos los engines y conjuntos existentes, generando sólo los nuevos bloques necesarios. Esto aumentará la cantidad de lógica ahorrada cuando se cumplen estas condiciones.

Para comprobar fácilmente qué optimizaciones se han hecho, podemos echar un vistazo a la consola (ver el entorno de desarrollo en la página 36), la cual nos muestra toda la información relevante sobre el proceso de generación de código VHDL:

Los engines son independientes en la implementación de Ignacio [14] y no se comunican entre ellos. Esta restricción se ha eliminado para poder permitir la compartición de hardware entre ellos.

Resultados experimentales

En esta sección se presentan los resultados experimentales obtenidos para varias colecciones de expresiones regulares.

Con el fin de obtener información precisa sobre dichas pruebas de rendimiento, se han incluido resúmenes del número de bloques generados, así como de la información relevante del reloj y del diseño del sistema obtenidos con la herramienta de Xilinx en el informe de síntesis.

Las pruebas de rendimiento se han realizado simulando una placa Virtex 6 XC6VLX75T utilizando los siguientes bancos de pruebas:

- Sitios web infectados por Lizamoon [48].
- Phishing para el banco Santander.
- Lista de direcciones IP infectadas [49].
- Conjunto de diversas direcciones IP.
- Conjunto de 2000 direcciones IP generadas aleatoriamente.

Para tener una idea experimental de la cantidad de LUTs utilizadas y la velocidad de reloj de la FPGA cuando se lleva a su límite de ocupación, se ha creado un pequeño programa para generar direcciones IP aleatorias con facilidad.

Work motivation

In computer science, pattern matching is the act of checking some sequence of tokens for the presence of the constituents of some pattern. The patterns generally have the form of either sequences or tree structures.

Nowadays, the main way to deal with regular expressions is using software solutions as is explained in the State of the Art section. This is very flexible, but suffers a lack of performance, and with the amount of patterns to recognize increasing every day, we need fast, simple and low-cost solutions for pattern recognition.

In this project a hardware solution is presented. This solution combines both the flexibility of software and the speed of hardware. The fact of combining the knowledge acquired in different areas during the studies makes this project quite different from the rest.

On one hand we have the necessity to deal with regular expressions, studied in Formal Languages and Automata Theory, combined with a good knowledge of Data Structures, Programming Methodology and Java programming. On the other hand we must know how to generate synthesizable VHDL code, so a good knowledge of Computer Architecture, Logic Design and Reconfigurable Hardware Design using FPGAs is needed.

Objectives

The main objective for this project is reducing the hardware generated to recognize regular expressions, maximizing the number of them processed by the FPGA.

As shown in figure 1, the code generator, which is written in Java, accepts a text file with sets of regular expressions, and then processes them, generating the VHDL file that describes the recognizer.

The recognizer proposed in this work has n outputs, one for each set. When a matching string from the input is detected, the associated output notifies it. This way, we can detect to which set a matching string belongs.

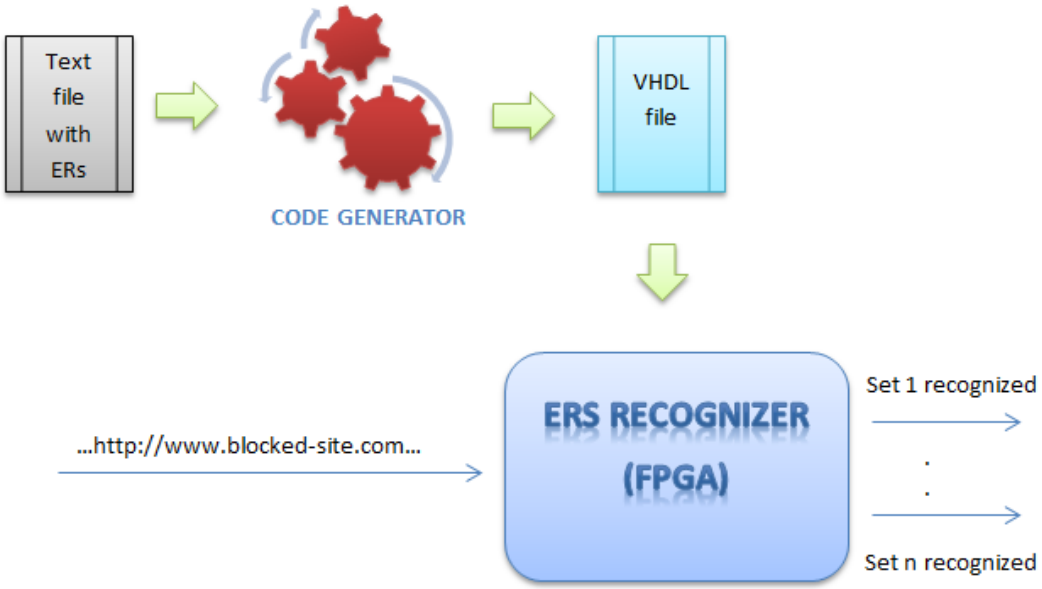

Figure 1 – Architecture diagram

The code generator is responsible of the hardware generation and minimization process. It has been achieved in two steps: first, complete engine reutilization, and second, partial engine reutilization.

Complete engine reutilization consists in reutilizing the same engine when a regular expression is used in multiple sets. This leads to great optimizations, sharing engines among sets without generating duplicated components.

Example:

SET1 $(jk)|(a(ab)*);\ (who)|(a((ab)*)d);\ ad;\ ftp;$
SET2 $ftp;\ http://;$ (1)

For example, the regular expression "ftp" seen in example 1 is present in both sets. Therefore, the set "SET2" uses the engine that recognizes "ftp" in "SET1" and does not generate an additional engine.

Partial engine reutilization is based on the same idea used for complete engine reutilization, but without limiting us to a complete regular expression. If a regular expression has a common part with any other one existing in any set, the hardware required for that part is not generated, and the existing logic is used instead (note that this has some limitations which is explained later). Thus we share hardware from every existing engine to any other, generating only the new logic needed. This boosts up the amount of saved hardware, due to the high number of optimizations made when these conditions are met.

In the following example, the subexpression "$http://www.$" is present at the start of two regular expressions in "SET1". Therefore, the hardware generated to recognize the second ER is only those needed to recognize "$web - site.org$", and the "$http://www.$" part will be taken from the first expression.

Example:

SET1 $http://www.example.com;\ http://www.web - site.org;$
SET2 $ftp;$ (2)

Creative process

To face the objectives proposed for this project, an exhaustive analysis of all the previous work has been made. This has involved studying the work of Sidhu and Prasanna [15] and Ignacio's engines-based implementation [14]. In addition, there has been a prior review of the state of the art to ensure that this is a good solution compared to others currently existing (see State of the Art section).

The research and development process has been split in several phases, ensuring that all the objectives for each deadline have been achieved, in a similar way to the Scrum [18] [19] process for software development used in Software Engineering (see figure 2).

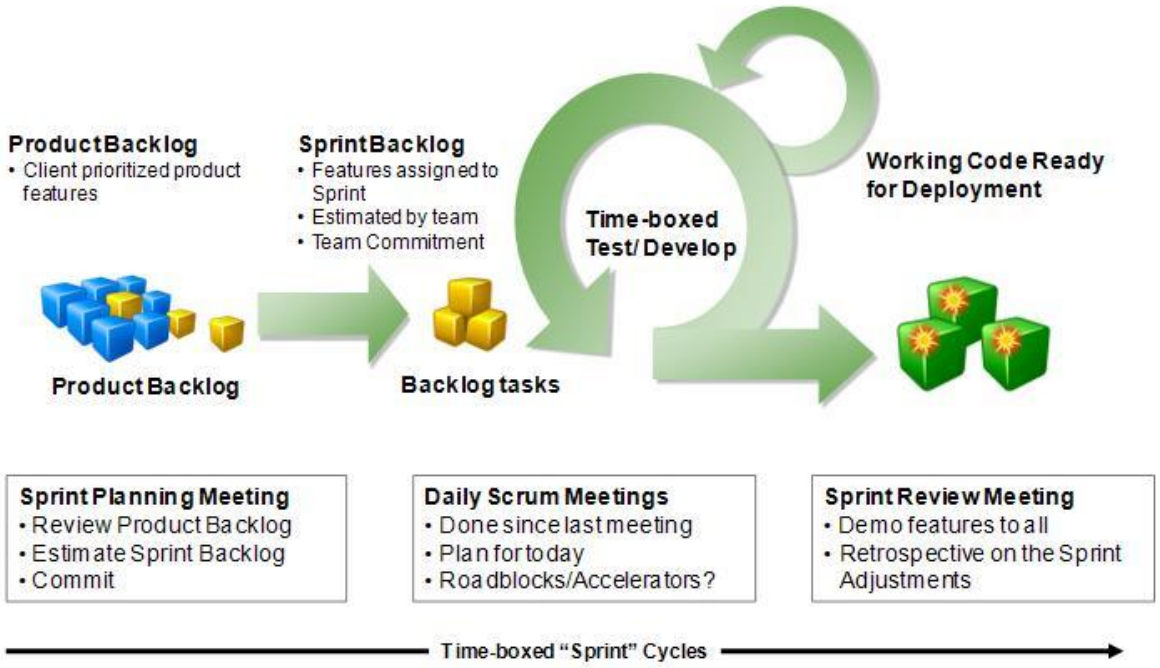

Figure 2 – The Scrum process

In each phase, several design alternatives have been considered, evaluating their simplicity, cost and implications in other modules and future upgrades. This makes of analysis a very important part of the development process, and although it consumes an important amount of time, also prevents future errors and simplifies the implementation phase.

A regular expression defines a search pattern for strings. This pattern may match one or several times or not for a given string. The abbreviation for regular expression is "regex". A simple example for a regular expression is a (literal) string. For example the regex "Hello World" will match exactly the phrase "Hello World".

The pattern defined by the regular expression is applied on the string from left to right. Once a source character has been used in a match, it cannot be reused. For example the regex "aba" will match "abababababa" only two times (aba_aba__).

Regular expressions can be used to search, edit and manipulate text and are used in several programming languages, e.g. Java, Perl, Groovy, etc. Unfortunately each language / program supports regex slightly different.

Specialized tools and utilities for working with regular expressions

These are some useful tools that let you work with regular expressions

- RegexBuddy [20] - Learn, create, understand, test, use and save regular expressions.
- RegexMagic [21] - Generate regular expressions using RegexMagic's patterns instead of the regular expression syntax.
- grep [22] - The utility from UNIX that first made regular expressions popular.
- PowerGREP [23] - Next generation grep for Microsoft Windows.

Programming languages and libraries

In order to work with regular expressions, the election of the programming language is a basis, as many provide many useful libraries and functions that make coding easier.

- Delphi [24] - Delphi XE and later ship with RegularExpressions and RegularExpressionsCore units that wrap the PCRE library. For older Delphi versions, you can use the TPerlRegEx component, which is the unit that the RegularExpressionsCore unit is based on.

- Gnulib [25] - Gnulib or the GNU Portability Library includes many modules, including a regex module.

- Groovy [26] - Groovy uses Java's java.util.regex package for regular expressions support. Groovy adds only a few language enhancements that allow you to instantiate the Pattern and Matcher classes with far fewer keystrokes.

- Java [27] - Java 4 and later include an excellent regular expressions library in the java.util.regex package.

- JavaScript [28] - If you use JavaScript to validate user input on a web page at the client side, using JavaScript's built-in regular expression support will greatly reduce the amount of code you need to write.

- .NET (dot net) [29] - Microsoft's new development framework includes a very powerful regular expression package, that you can use in any .NET-based programming language such as C# (C sharp) or VB.NET.

- PCRE [30] - Popular open source regular expression library written in ANSI C that you can link directly into your C and C++ applications, or use through a .so (UNIX/Linux) or a .dll (Windows).

- Perl [31] - The text-processing language that gave regular expressions a second life, and introduced many new features. Regular expressions are an essential part of Perl

- PHP [32] - Popular language for creating dynamic web pages, with three sets of regex functions.

- POSIX [33] - The POSIX standard defines two regular expression variants that are implemented in many applications, programming languages and systems.

- PowerShell [34] - Windows PowerShell is a programming language from Microsoft that is primarily designed for system administration. PowerShell can access the .NET Regex classes directly.

- Python [35] - Popular high-level scripting language with a comprehensive built-in regular expression library

- R [36] - The R Language is the programming languages used in the R Project for statistical computing. It has built-in support for regular expressions based on POSIX and PCRE.

- REALbasic [37] - Cross-platform development tool similar to Visual Basic, with a built-in RegEx class based on PCRE.

- Ruby [38] - Another popular high-level scripting language with comprehensive regular expression support as a language feature.

- Tcl [39] - Tcl, a popular "glue" language, offers three regex variants. Two POSIX-compatible variants and an "advanced" Perl-style variant.

- VBScript [40] - Microsoft scripting language used in ASP (Active Server Pages) and Windows scripting, with a built-in RegExp object implementing the regex variant defined in the JavaScript standard.

- Visual Basic 6 [41] - Last version of Visual Basic for Win32 development. You can use the VBScript RegExp object in your VB6 applications.

- wxWidgets [42] - Popular open source windowing toolkit. The wxRegEx class encapsulates the "Advanced Regular Expression" engine originally developed for Tcl.

- XML Schema [43] - The W3C XML Schema standard defines its own regular expression variant for validating simple types using pattern facets.

- XQuery and XPath [44] [45] - The W3C standard for XQuery 1.0 and XPath 2.0 Functions and Operators extends the XML Schema regex variant to make it suitable for full text search.

Databases

Modern databases often offer built-in regular expression features that can be used in SQL statements to filter columns using a regular expression. With some databases you can also use regular expressions to extract the useful part of a column, or to modify columns using a search-and-replace.

- MySQL [46] - MySQL's REGEXP operator works just like the LIKE operator, except that it uses a POSIX Extended Regular Expression.

- Oracle [47] - Oracle Database 10g adds 4 regular expression functions that can be used in SQL and PL/SQL statements to filter rows and to extract and replace regex matches. Oracle implements POSIX Extended Regular Expressions.

- PostgreSQL [48] - PostgreSQL provides matching operators and extraction and substitution functions using the "Advanced Regular Expression" engine also used by Tcl.

R. Sidhu and V.K. Prasanna's algorithm

Sidhu and Prasanna present an efficient method for using FPGAs for fast regular expression matching, finding all strings in given text that match the regular expression [15].

Their algorithm improves memory costs significantly: while a serial machine requires $O(2^n)$ memory and takes $O(1)$ time per character to match a regular expression of length n, their proposed approach require $O(n^2)$ space and still processes a text character in $O(1)$ time (one clock cycle). This is the first use of a nondeterministic state machine on programmable logic.

The approach proposed by them constructs a Nondeterministic Finite Automaton (NFA) and uses it to process text characters. The algorithm for NFA construction is linear in time and quadratic in space in the length of the regular expression. The NFA can process one text character every clock cycle. To process a text character in constant time on a serial machine, the construction of a Deterministic Finite Automata (DFA) is required. DFA construction requires, in the worst case, time and space exponential on the length of the regular expression. Their proposed approach drastically reduces time and space requirements by exploiting the reconfigurability and parallelism of FPGAs.

Previous work using FPGAs for string matching has primarily focused on searching for a specific string [12] [13]. Attempts at regular expression matching using FPGAs have been based on DFAs and suffer from the same inefficiencies as serial machine implementations like grep.

Sidhu and Prasanna also describe the logic structures needed to implement the NFA associated to a regular expression. These structures are:

- Single character/Basic block: The output is 1 only when the flip-flop stores a 1 and the input matches the character stored in the inside comparator (figure 3.a).

- $r_1|r_2$: The only logic required is an OR gate to combine the outputs from N_1 and N_2 (figure 3.b).

- $r_1\, r_2$: The only logic required is just three wires (figure 3.c).

- $r1^*$: An OR gate is used to combine the two inputs to the initial state of N_1. Another OR gate combines the two inputs to generate the accept output (figure 3.d).

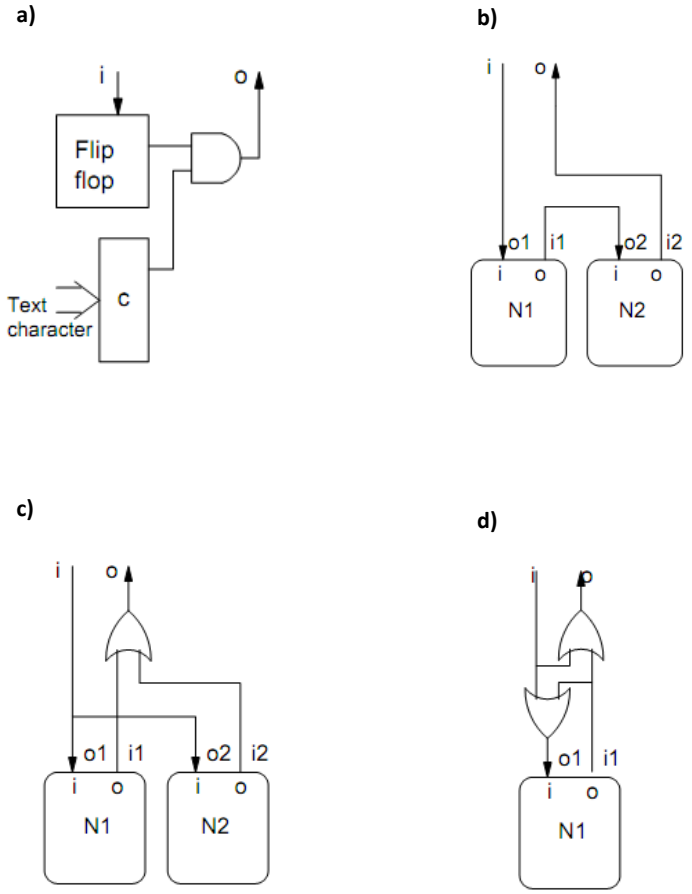

Figure 3 - Diagrams for blocks: a) Single character/basic block, b) $r_1|r_2$, c) $r_1\, r_2$, and d) $r1^*$

The logic structures shown in figure 3 can be combined to construct an NFA for a given regular expression. In order to recognize it, we need to connect these blocks properly, activating them only when necessary. When a single character block B_i is active, it will propagate the result of comparing the input character i and its matching character to the next block B_{i+1}, activating it when the result of the comparison is true. In case that the block B_i implements a final state of the automata, the comparison result itself will be its output.

```
for (i=1; i<regexp_len; i++){
        switch (regexp[i]){
                case char:      place_char(regexp[i],&p);
                                push(p);
                case |:         place_|(&p);
                                pop(&p1);
                                route1(p,p1);
                                pop(&p2);
                                route2(p,p2);
                                push(p2);
                case •:         place_•(&p);
                                pop(&p1);
                                route1(p,p1);
                                pop(&p2);
                                route2(p,p2);
                                push(p2);
                case *:         place_*(&p);
                                pop(&p1);
                                route1(p,p1);
                                push(p);
                }
}
route_input_high(p);
route_output_ff(p);
```

Algorithm 1 - Sidhu and Prasanna's algorithm

The Sidhu and Prasanna's algorithm (see algorithm 1) for NFA construction accepts the regular expression in postfix form (due to efficiency reasons), and uses a stack and the following subroutines: "$place\_char$", "$place\_|$", "$place\_.$" and "$place\_*$", which respectively place the logic structures for a character, and the metacharacters "|", "$*$" and ".". All the subroutines return a pointer (p) to the created structure. "$route1$" and "$route2$" create bidirectional connections between the logic structures given by parameter: $route1(p,q)$ connects the o output of q to the $i1$ input of p and the $o1$ output of p to the i input of q. Similarly, $route2(p,q)$ creates connections to $i2$ and $o2$ ports of p. Finally, the algorithm connects i and o ports of the logic structure for the last symbol in the regular expression.

Ground work

The work presented in this article is based on the previous work done by Ignacio Martín Santamaría. He made the first Java implementation of R. Sidhu and V.K. Prasanna's algorithm presented in "Fast Regular Expression Matching Using FPGAs" (FCCM'01) [14] and under the direction of Marcos Sánchez-Élez Martín.

The main difference between this architecture and others is the use of a code generator. This idea makes the program management very easy due that VHDL does not have the same flexibility and modularity as Java or any other similar high-level programming language. Moreover, a software implementation of the generator gives the possibility to make a comprehensive analysis to generate highly optimized VHDL code.

A text file with regular expressions is taken by the code generator, which is a Java program that implements Sidhu and Prasanna's algorithm and generates a VHDL file, which describes the architecture that recognizes the regular expressions from the text file.

The target is, given a continuous flow of characters, detect the presence of any subchain that matches any of the supported regular expressions. Each regular expression is implemented individually by a specific recognizer. Thus, there are as many recognizers as the number of them supported. These recognizers are called

engines, and each one will be able to recognize one regular expression. Once configured, it's not possible change the regular expression the engine recognizes. The complexity of each engine is determined by the length and structure of the ER that defines it.

Every engine synchronously receives a character flow as input, each one represented by one byte, and processes it notifying with a bit at the output when a match occurs between the processed chain and the regular expression it recognizes.

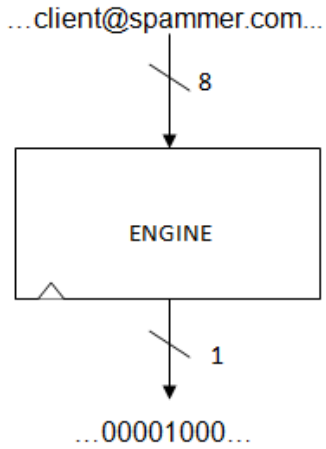

Figure 4 - Engine diagram

In this version, each engine works independently and does not need any type of communication with the rest.

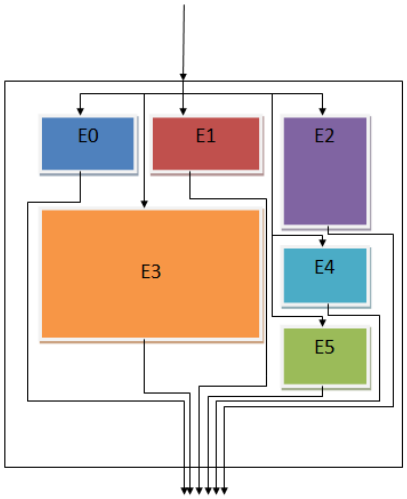

Figure 5 - A possible implementation

Sidhu and Prasanna's algorithm and Ignacio's implementation have some limitations with that it would be impossible to achieve the desired optimizations without changes. These changes involve engines connections and how their internal blocks are connected.

As commented on the introduction (page 18), partial engine reutilization (see page 50) consists in reutilizing the same engine when parts of a regular expression are used in multiple sets. This cannot be done without modifying Sidhu and Prassana's algorithm.

It might not be needed generating all the blocks for a regular expression, as a prefix of it may have been previously generated. In this case, we use the output signal associated with the last block of the common part rather than replicating its components. To achieve this, apart from modifying Sidhu's algorithm, we must support communication between engines, eliminating the restriction imposed by Ignacio where each engine acts independently from the rest.

What is a FPGA?

A Field Programmable Gate Array (FPGA) is an integrated circuit that can be programmed (with HDL) to become almost any kind of digital circuit. FPGAs consist of array of programmable logic blocks of different types surrounded by routing fabric that allows blocks to be programmably interconnected. The array is surrounded by input/output blocks.

Before the advent of programmable logic, custom logic circuits were built at the board level using standard components, or at the gate level in expensive application-specific (custom) integrated circuits. A FPGA contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. A user's design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells and interconnections form a fabric of basic building blocks for logic circuits. Complex designs are created by combining these basic blocks to create the desired circuit.

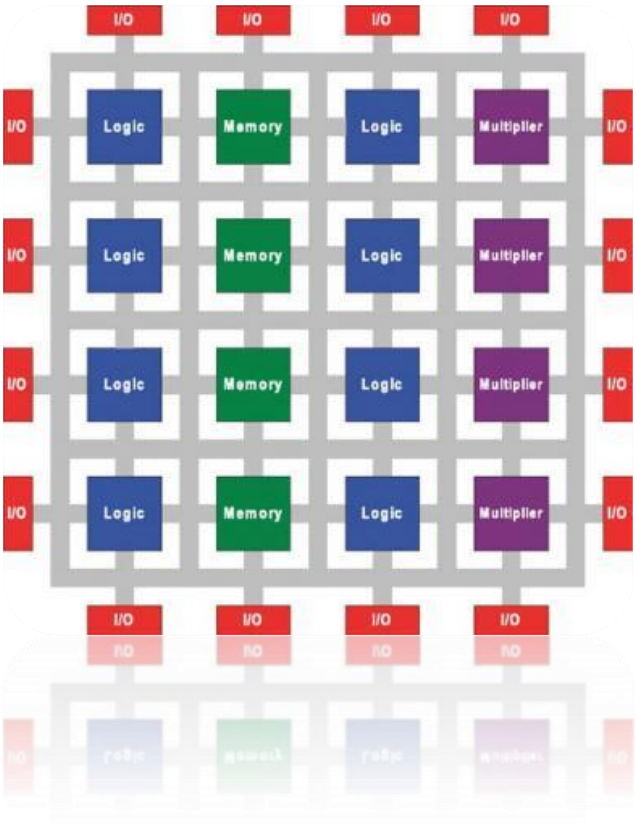

Figure 6 - Basic architecture of a FPGA

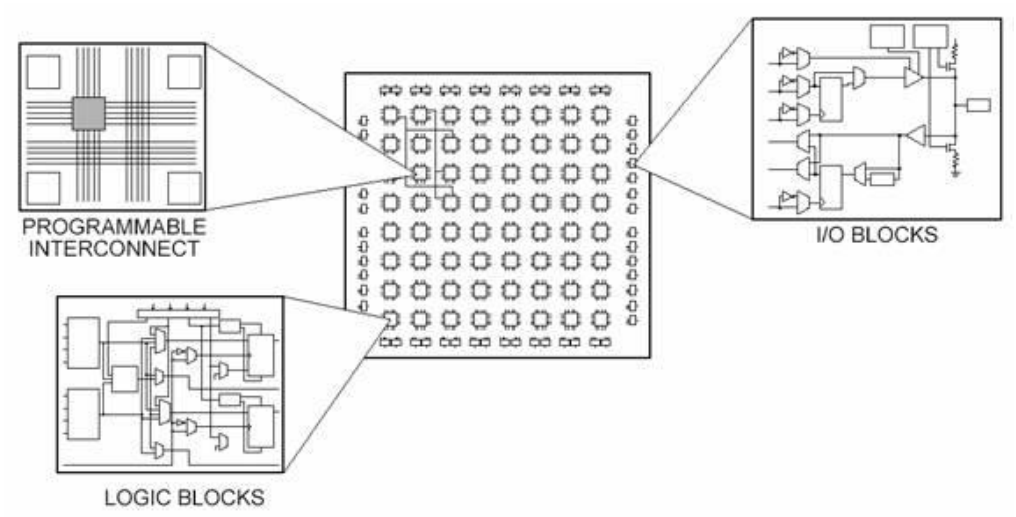

Figure 7 - Different parts of an FPGA

What does a logic cell do?

The logic cell architecture varies between different device families. Generally speaking, each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a Boolean logic function specified in the user program. In most families, the user also has the option of registering the combinatorial output of the cell, so that clocked logic can be easily implemented. The cell's combinatorial logic may be physically implemented as a small look-up table memory (LUT) or as a set of multiplexers and gates. LUT devices are a bit more flexible and provide more inputs per cell than multiplexer cells at the expense of propagation delay.

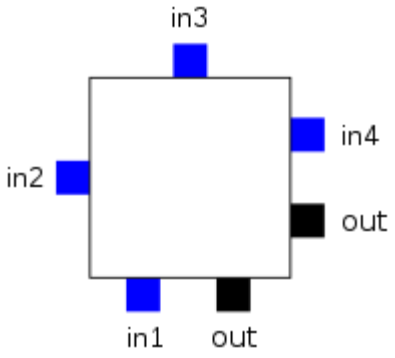

Figure 8 - Logic block pin locations

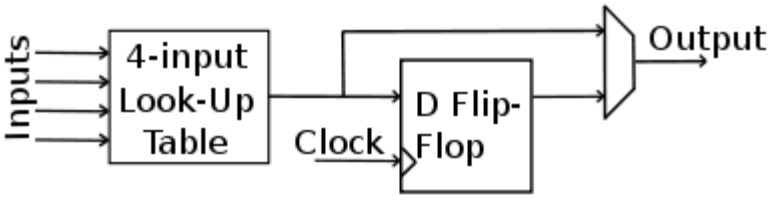

Figure 9 - Typical logic block

So what does 'Field Programmable' mean?

Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. A typical integrated circuit performs a particular function defined at the time of manufacture. In contrast, the FPGA's function is defined by a program written by someone other than the device manufacturer. Depending on the particular device, the program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up. This user programmability gives the user access to complex integrated designs without the high engineering costs associated with application specific integrated circuits.

And how are FPGA programs created?

Figure 10 - Virtex 5 family FPGA board

Individually defining the many switch connections and cell logic functions would be a daunting task. Fortunately, this task is handled by special software. The software translates a user's schematic diagrams or textual hardware description language code and then places and routes the translated design. Most of the software packages have options to allow the user to configure implementation, placement and routing to obtain better performance and utilization of the device. Libraries of more complex function macros (e.g. Adders, Multipliers, Multiplexers, etc.) further simplify the design process by providing common circuits that are already optimized for speed or area.

However, not every HDL description is efficiently synthesizable, as it depends on the tools associated to the specific platform.

Introduction

In this chapter the development environment used is shown, highlighting the different tools used along the process of development and testing.

As mentioned before, this project combines the flexibility of software with the speed of hardware, so we have to choose a powerful and easy to use development environment for both parts.

In our case, due to the complexity of this project, the development, debugging and testing process have been divided into several parts, with special emphasis on the modularity of each.

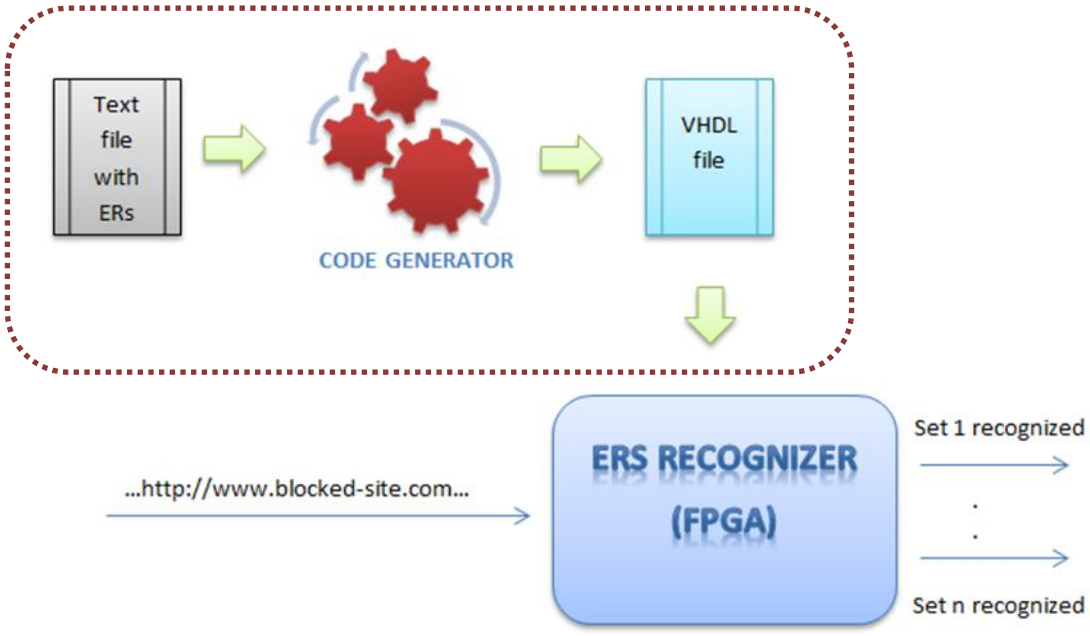

Figure 11 – Architecture diagram

As shown in figure 11, the code generator, which is written in Java, accepts a text file with sets of regular expressions, and then processes them, generating the VHDL file that describes the recognizer.

The VHDL file can be used with any software tool for synthesis, analysis and simulation of VHDL designs. This allows:

- Check for syntax errors in the VHDL code.
- View the RTL/Technology schematic of the design.
- Simulate the design for a specific board (e.g., Virtex 5, Spartan 3, etc.).
- Obtain the Timing Summary (e.g., minimum clock period, maximum combinational path delay, etc.).
- Calculate the amount of logic generated.
- View the amount of resources used of the FPGA.
- Generate the programming file used for configuring the target device.

Eclipse IDE [50] and the Java language have been used for programming the VHDL code generator. On the other hand, Xilinx ISE [51] has been used for debugging and testing the generated VHDL code. Moreover, Xilinx ISim and Modelsim have been used for simulating the design.

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It can be used to develop applications in Java and other programming languages.

Xilinx ISE is a design tool for synthesis and analysis of HDL which lets you configure the destination board in order to simulate and synthetize the design properly. This is very helpful because it allows doing both tasks with ease and generates the programming file that can be downloaded into the FPGA.

Below can be seen a detailed description of the most relevant parts of the development environment: input file with regular expressions, structure of the code generator and testing and debugging process.

Input file structure

The code generator accepts multiple sets of regular expressions from a text file. This file uses the following format:

- Each line represents a set of regular expressions, being the first word of the line the name of the set.
- The text file can be interpreted as a database of regular expressions, where they are classified by its type (name of the set).
- There is virtually no limit to the number of existing sets, neither the expressions contained in each.
- The list of regular expressions belonging to a set is described after its name, leaving a white space after it.
- All the regular expressions end with a semicolon and are described in infix form.

In figure 12 is shown an example of text file containing eighteen sets with multiple regular expressions.

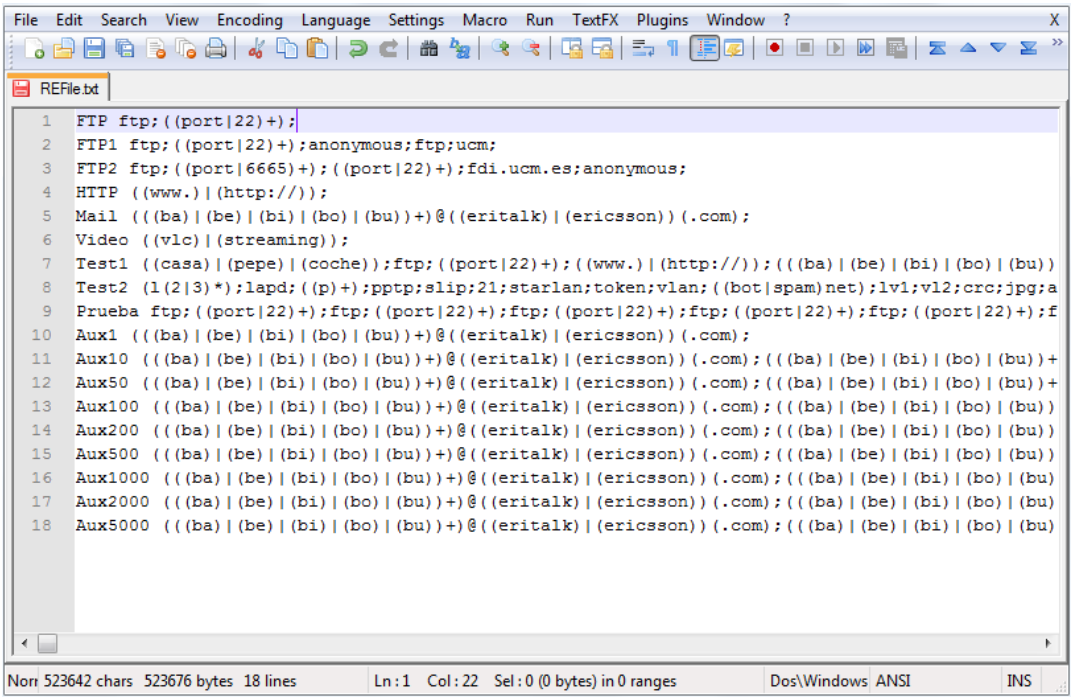

Figure 12 – Input file with regular expressions

Code generator

The code generator is a program written in Java that takes various sets of regular expressions. To easily generate recognizers, the program accepts the name of the sets (given by parameter) and a string of characters to simulate the input flow. The string of characters given by parameter is needed to dynamically generate another VHDL files used for simulation. Thus we can generate recognizers in a fast and easy way.

There are three VHDL files generated by the code generator

- VHDLFile.vhd: Main VHDL file that describes the recognizer.

- reconocedorERS.vhd: Top level module generated for simulation purposes. It uses the recognizer described in VHDLFile.vhd.

- reconocedorERS_FPGA.vhd: Top level module generated for downloading and testing the recognizer into a FPGA. This module uses the recognizer described in VHDLFile.vhd.

The code generator is composed of six classes:

- Cadenas: This class contains the regular expressions and subexpressions in postfix form, the name of signals of the engines and the necessary methods used to detect common subexpressions.

- FilehanderHander: A class used to read the input text file.

- Main: Main class of the program.

- Parser: Class used to deal with regular expressions. Its main feature is obtaining the postfix form of a given regular expression.

- Stack: A generic stack used to implement Sidhu's algorithm.

- VHDLGenerator: This class generates the VHDL code.

Debugging and testing

The debugging and testing is an essential part of the development process. All the relevant information about this part is shown below.

Simulation

As explained before, the string of characters given by parameter is needed to dynamically generate VHDL files used for simulation. This string will simulate the input flow of characters during simulation. This way, the top level module containing the recognizer is dynamically generated to store this string and parse a char of it to the recognizer each clock cycle.

Moreover, the code generator can dynamically generate two different simulation files: “reconocedorERS.vhd” and “reconocedorERS_FPGA.vhd”. The main difference between them is the use of a clock divider to visually see the output on the FPGA for a long period of time when a match occurs. During the testing for this project, the system outputs are connected to the Spartan 3 LEDs, which allows verifying the simulation behavior.

Along the testing of this project, all the debugging process has been done using behavioral simulations with the file “reconocedorERS.vhd” in first place, and post-route simulations using the file “VHDLFile.vhd” once the behavioral is passed (see experimental results on page 61).

The tools used for simulation testing are Xilinx ISim and Modelsim. Once verified the correct behavior of the design (see figures 13 and 14), then it is downloaded into the FPGA board (a Spartan 3 is our case) using the file “reconocedorERS_FPGA.vhd”.

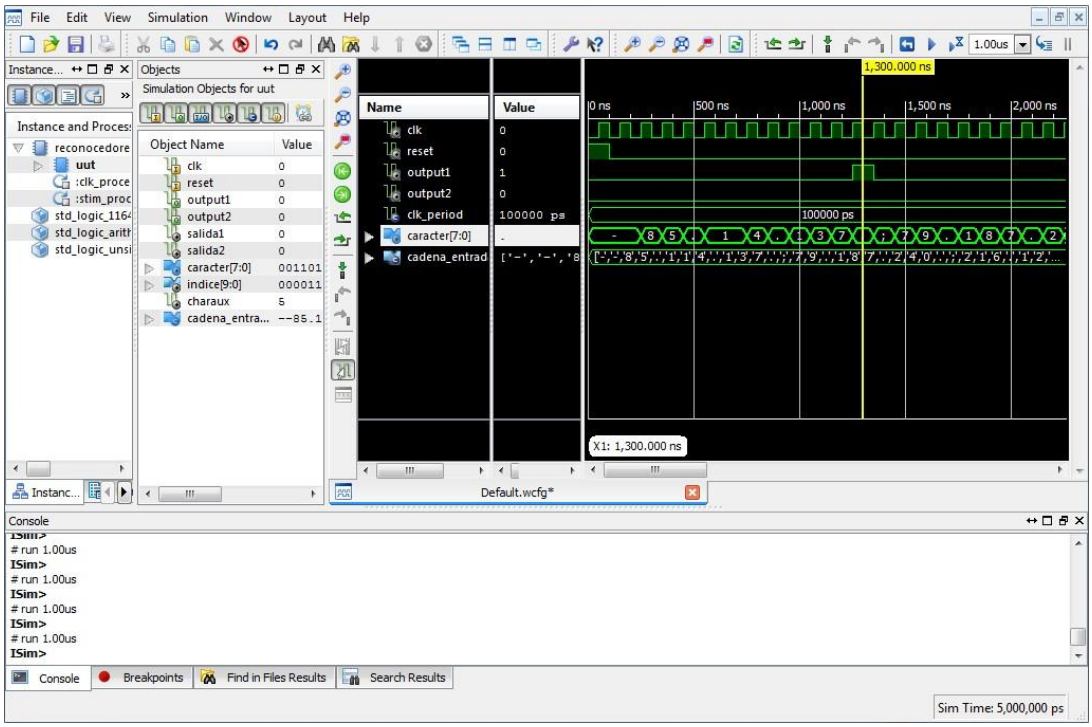

Figure 13 - Behavioral simulation results using Xilinx ISim

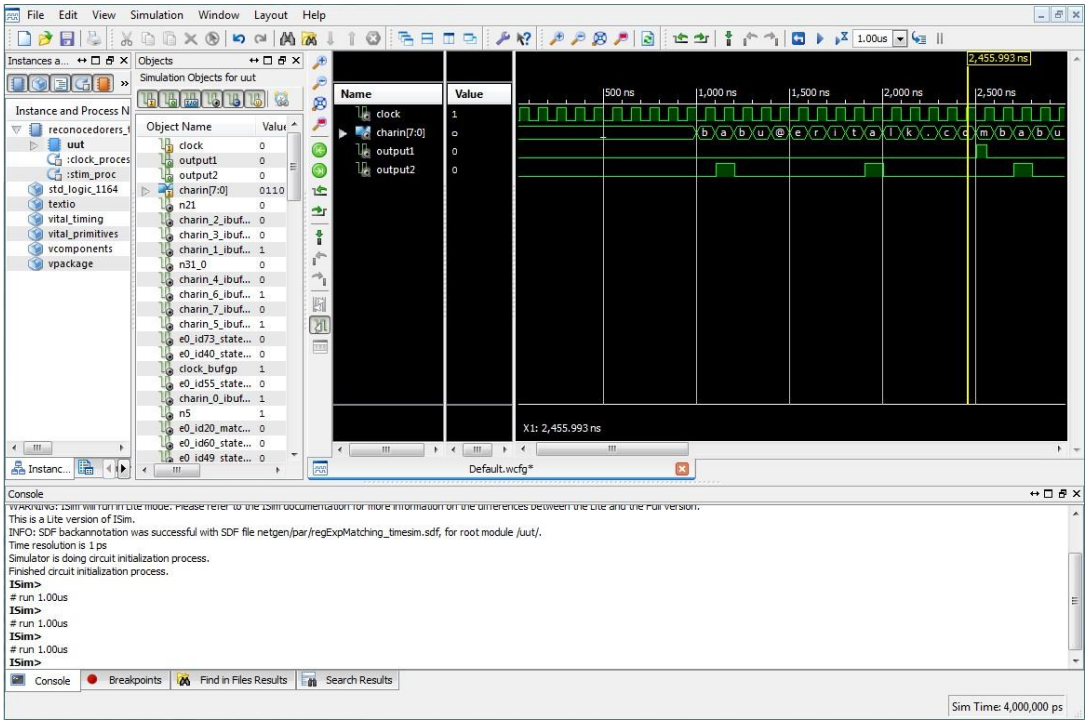

Figure 14 - Post-Route simulation

In order to be able to simulate our designs using Xilinx ISim and Modelsim, we need VHDL test benches. These are VHDL files created only for simulation purposes that give the simulator all the necessary information about the design and the stimulus applied to the input signals (e.g. clock and input character in our case).

The test benches used for simulation in this project are:

- reconocedorERS_Test_Bench.vhd: Test bench used for standard simulation.

- reconocedorERS_Test_Bench_Benchmark.vhd: Test bench specifically designed for FPGA simulation. In this version only the main module described in the “VHDLfile.vhd” file is used and the input characters are manually given to the input ports each clock cycle. This way, the extra hardware added in the regular design used for debugging is removed in order to get realistic post route simulations (see experimental results on page 61).

Console information

During debugging, all the relevant information about modules generation and regular expressions optimizations is essential. Therefore, the code generator has been modified to show all this information on the console.

As can be seen in the figure 15, the console shows the components generated for each expression, along with its postfix form and the optimizations made.

First, the console shows the relevant information of the regular expressions optimization process for each set. It gives the list of the regular expressions in postfix form that belong to each set, besides the subexpression reused below each (if any).

In case of existing a reused subexpression for a regular expression, the console shows the engine it belongs, along with the name of the signal associated to its output. In other case, it will show “No subexpressions have been reused”.

Below this information, comes the engines information section, denoted by the line “***ENGINES COMPONENTS***”. In this section are shown all the engines generated by the code generator for each regular expression, along with their contents (‘*basic' block as BB*, ‘*star’ block as STAR*, ‘*and’ block* as AND and ‘*or’* block as OR).

As can be seen in the figure 15, each information block of each engine begins with the line “ENGINE X from SET Y”, being the numeration of the engines consecutive. Below this line is showed the regular expression in postfix form recognized by the engine, followed by the blocks generated for it along with their identifications.

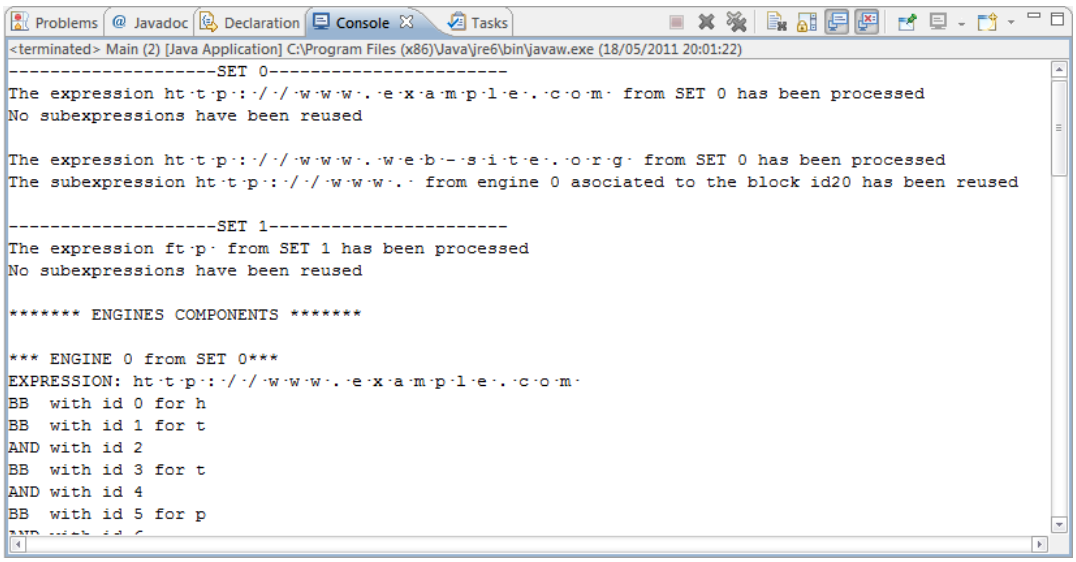

Figure 15 - Console information

For example, the line below "ENGINE 0 FROM set 0" showing "BB with id3 for "*t*" means that the "*t*" symbol of the regular expression generates a "*basic block*" with id "*id3*" for this engine. The description of the different logic structures designed by R. Sidhu and V.K. Prasanna's can be seen in page 28. How these blocks are generated depends on the structure of each regular expression and the kind of optimizations performed (see complete and partial engine reutilization in pages 48 and 50).

The following table shows the equivalences between these blocks, the postfix symbols of a regular expression and the abbreviation shown in the console.

Postfix symbol	**Type of block**	**Console abbreviation**
Simple character	*Single character / Basic Block*	*BB*
|	$r_1 \| r_2$	*OR*
•	$r_1\ r_2$	*AND*
*	$r1^*$	*STAR*

Introduction

As stated in the objectives in page 18, the first modification made on Ignacio's work is adding support for multiple sets recognition. Once done this upgrade, takes place the hardware optimization process, composed of two parts: complete and partial engine reutilization (see pages 48 and 50).

Therefore, there are as many outputs as sets to be able to detect to which set each regular expression belongs.

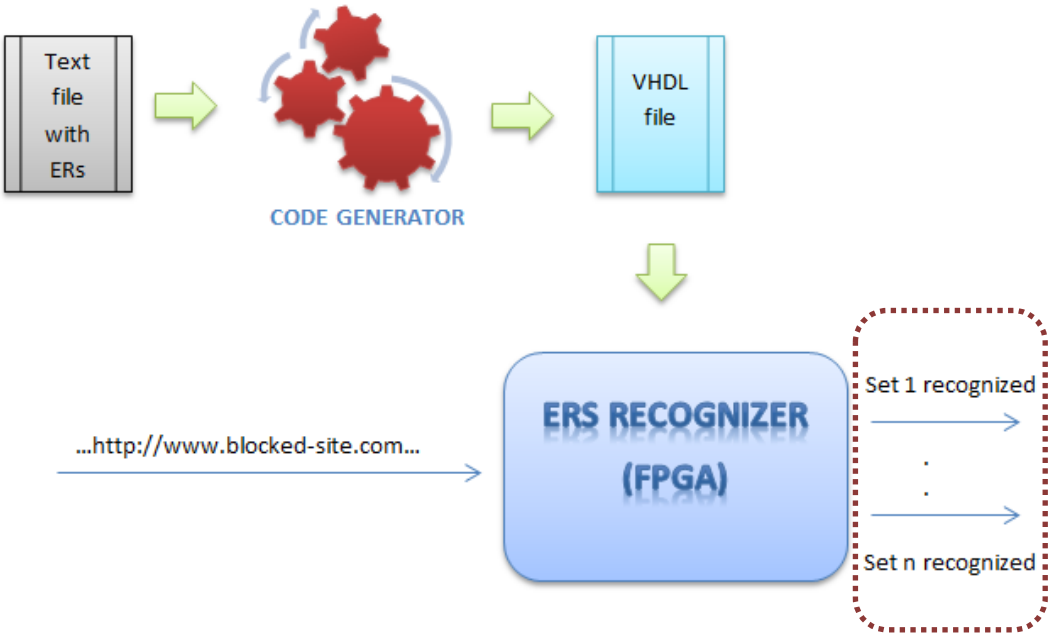

Figure 16 – Architecture diagram

As shown in figure 16, the recognizer has n outputs, one for each set. When a matching string from the input is detected, the associated output notifies it. The figure 17 shows the top level schematic for a recognizer of two outputs.

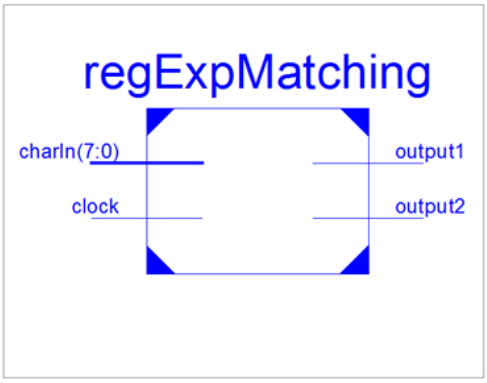

Figure 17 - Top level module

Implementation

In order to achieve the desired functionality, we need to know which regular expressions belong to each set (we distinguish two sets in this version). Therefore, we read two lines with regular expressions from the text input file, one for each set.

Example:

SET1 $a(aba)*;((aa)|(bb)|(cc));(bba)|(c);$
SET2 $ftp;$ (3)

If we take a look at example 3, we can see that the input text file is structured in lines (sets of regular expressions), starting each line with the name of the set, and followed by all its regular expressions separated by semicolons.

In this first modification, the recognizer is structured in two parts; each dedicated to recognize one set, and does not share any logic.

In the next section is explained the first part of the hardware optimization process: complete engine reutilization.

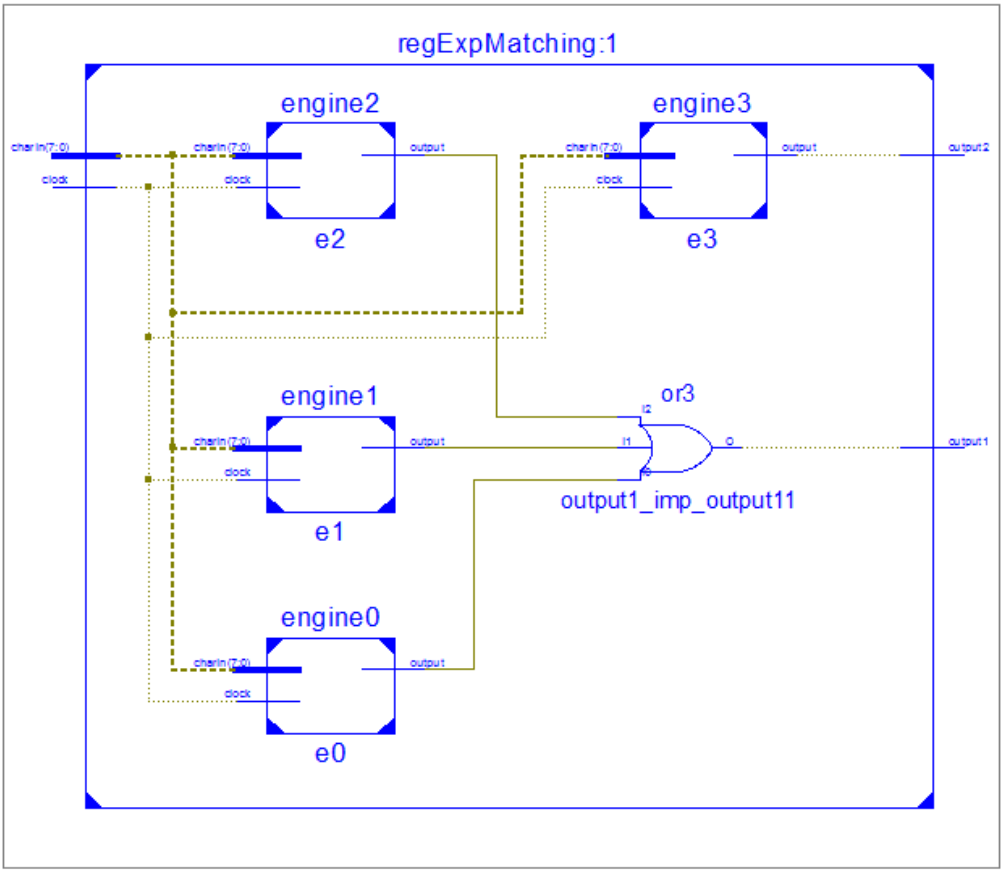

Figure 18 - Hardware generated for example 3

COMPLETE ENGINE REUTILIZATION

Once added support for multiple sets, we want to reuse all the logic generated to recognize a regular expression of a set in another one instead of replicating it. This is the first enhancement implemented in order to reuse hardware.

Complete engine reutilization consists in using an existing engine when a regular expression of a set is contained in another set without replicating engines. This will lead to great improvements in area when the same regular expression is used in several sets.

Example:

SET1 $a(aba)*;((aa)|(bb)|(cc));(bba)|(c);ftp;$
SET2 $ftp;$ (4)

The regular expression "ftp" is present in both sets. Therefore, the set "SET2" uses the engine that recognizes "ftp" in "SET1" and does not generate an additional identical engine.

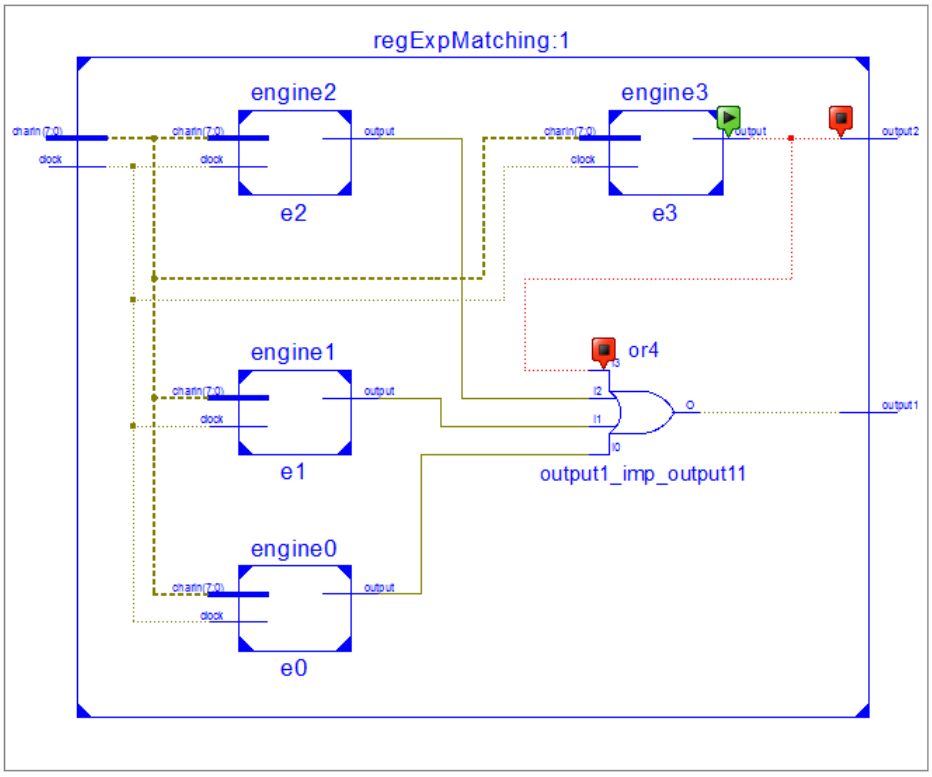

Figure 19 - Hardware generated for example 4

As we can see, there are four different regular expressions in total ("ftp" is repeated), so four engines have been generated. As shown in figure 19, the engine3 is being used by both outputs instead of having two different engines recognizing the same regular expression "ftp".

The engine "e0" recognizes $a(aba)*$, "e1" recognizes $((aa)|(bb)|(cc))$, "e2" $(bba)|(c)$ and engine "e3" ftp.

This process is achieved comparing all the postfix forms of the regular expressions of each set to the rest. If a regular expression has been previously generated, then the output of the existing engine is connected to the outputs of the current set, without replicating engines.

Introduction

This phase is quite more complicated to implement than the rest, as there are some limitations on Sidhu and Prasanna's algorithm and Ignacio's original implementation (see restrictions in page 60), which affect to the number of optimizations that can be made.

Most of these limitations have been solved, as is explained along this chapter, allowing us to save hardware (see experimental results in page 61).

As commented before, partial engine reutilization is based on the same idea used for complete engine reutilization, but without limiting us to a complete regular expression. When one expression has a common part with any other one existing in any set, we do not generate the hardware required for that part, and use the existing logic previously generated instead. Therefore, we share hardware from every existing engine to any other, generating only the new logic needed. This will boost up the amount of saved hardware when these conditions are met.

Example:

SET1 $http://www.example.com;\ http://www.web-site.org;$
SET2 $ftp;$ (5)

In this case, the subexpression "$http://www.$" is present at the start of two regular expressions in "SET1". Therefore, the hardware generated to recognize the second ER is only those needed to recognize "$web-site.org$" and the "$http://www.$" part is taken from the first expression.

To easily check what optimizations have been made, we can take a look at the console (see development environment in page 36), which shows the following information (figures 20 and 21):

```
    --------------------SET 0-----------------------
    The expression ht·t·p·:·/·/·w·w·w·.·e·x·a·m·p·l·e·.·c·o·m· from SET 0 has been
processed
    No subexpressions have been reused

    The expression ht·t·p·:·/·/·w·w·w·.·w·e·b·-·s·i·t·e·.·o·r·g· from SET 0 has been
processed
    The subexpression ht·t·p·:·/·/·w·w·w·.· from engine 0 associated to the block
id20 has been reused

    --------------------SET 1-----------------------
    The expression ft·p· from SET 1 has been processed
    No subexpressions have been reused

    ******* ENGINES COMPONENTS *******

    *** ENGINE 0 from SET 0***
    EXPRESSION: ht·t·p·:·/·/·w·w·w·.·e·x·a·m·p·l·e·.·c·o·m·
    BB  with id 0 for h
    BB  with id 1 for t
    AND with id 2
    BB  with id 3 for t
    AND with id 4
    BB  with id 5 for p
    AND with id 6
    BB  with id 7 for :
    AND with id 8
    BB  with id 9 for /
    AND with id 10
    BB  with id 11 for /
    AND with id 12
    BB  with id 13 for w
    AND with id 14
    BB  with id 15 for w
    AND with id 16
    BB  with id 17 for w
    AND with id 18
    BB  with id 19 for .
    AND with id 20
    BB  with id 21 for e
    AND with id 22
    BB  with id 23 for x
    AND with id 24
    BB  with id 25 for a
    AND with id 26
    BB  with id 27 for m
    AND with id 28
    BB  with id 29 for p
    AND with id 30
    BB  with id 31 for l
    AND with id 32
    BB  with id 33 for e
    AND with id 34
    BB  with id 35 for .
    AND with id 36
    BB  with id 37 for c
    AND with id 38
    BB  with id 39 for o
    AND with id 40
    BB  with id 41 for m
    AND with id 42
```

Figure 20 - Optimization results for example 6, part one

```
*** ENGINE 1 from SET 0***
EXPRESSION: ht·t·p·:·/·/·w·w·w·.·w·e·b·-·s·i·t·e·.·o·r·g·
BB  with id 21 for w
AND with id 22
BB  with id 23 for e
AND with id 24
BB  with id 25 for b
AND with id 26
BB  with id 27 for -
AND with id 28
BB  with id 29 for s
AND with id 30
BB  with id 31 for i
AND with id 32
BB  with id 33 for t
AND with id 34
BB  with id 35 for e
AND with id 36
BB  with id 37 for .
AND with id 38
BB  with id 39 for o
AND with id 40
BB  with id 41 for r
AND with id 42
BB  with id 43 for g
AND with id 44

*** ENGINE 2 from SET 1***
EXPRESSION: ft·p·
BB  with id 0 for f
BB  with id 1 for t
AND with id 2
BB  with id 3 for p
AND with id 4
```

Figure 21 - Optimization results for example 6, part two

For example 5 there are three engines, one for each regular expression, but all the blocks present in engine 1 with ids from 0 to 19 are not present in engine 1: we can read in the in the fifth line of the console that the subexpression "$ht \bullet t \bullet p \bullet :\bullet/\bullet/\bullet w \bullet w \bullet w \bullet.\bullet$" from engine 0 associated to the block id20 has been reused. This means that the engine 1 connects its input to the output associated to the subexpression "$ht \bullet t \bullet p \bullet:\bullet/\bullet/\bullet w \bullet w \bullet w \bullet.\bullet$" from engine 0, which is associated to the block id20.

```
..
BB  with id 0 for h
BB  with id 1 for t
AND with id 2
BB  with id 3 for t
AND with id 4
BB  with id 5 for p
AND with id 6
BB  with id 7 for :
AND with id 8
BB  with id 9 for /
AND with id 10
BB  with id 11 for /
AND with id 12
BB  with id 13 for w
AND with id 14
BB  with id 15 for w
AND with id 16
BB  with id 17 for w
BB  with id 19 for .
..
```

Figure 22 - Blocks not present in engine 2

The engines were originally independent in Ignacio's implementation and did not communicate among them (see previous work in page 30). This restriction has been eliminated to allow hardware sharing between engines.

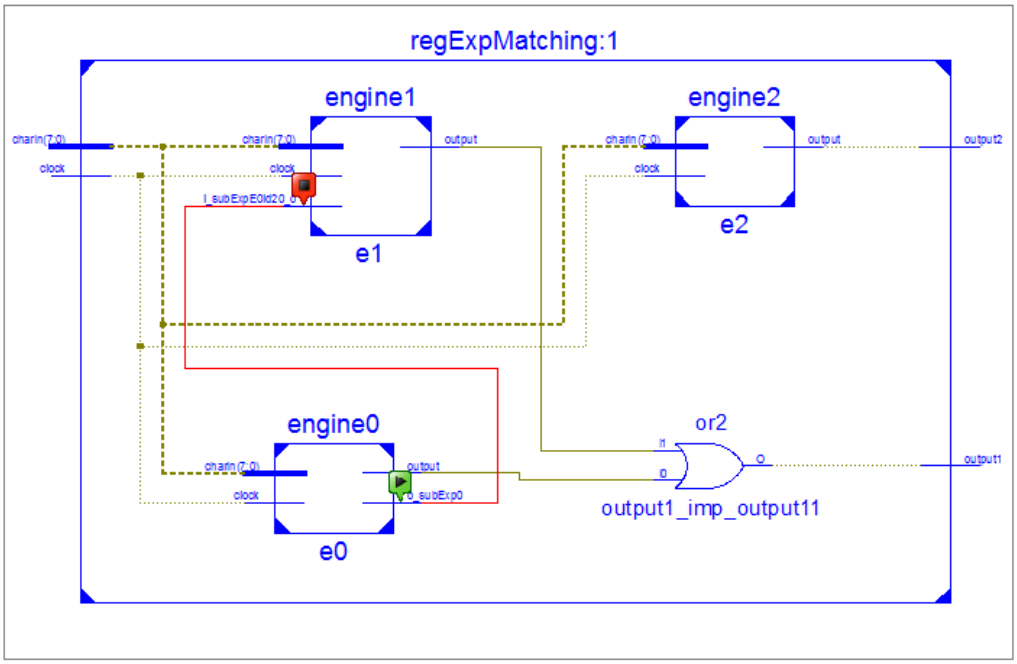

Figure 23 - External engine connections for example 5

As shown in figure 23, the engine 1 has an additional input, connected to the new output of the engine 0. This signal activates when the subexpression "$ht \bullet t \bullet p \bullet{:}\bullet/\bullet/\bullet w \bullet w \bullet w \bullet{.}\bullet$" is recognized by engine 0, and then begins the matching for the rest of the expression: "$w \bullet e \bullet b \bullet - \bullet s \bullet i \bullet t \bullet e \bullet{.}\bullet o \bullet r \bullet g \bullet$".

Regular expressions optimization

This section explains the algorithm used for partial engine reutilization and its limitations. This algorithm performs an exhaustive analysis of each regular expression of the input sets, looking for common prefixes between them in order to share hardware among engines.

A pair of regular expressions must have a common part starting from their left side (in postfix mode) to allow component sharing. This means that the engine that recognizes one of the regular expressions has an extra input that activates when the common part is recognized by the other engine, so another signal is generated to connect these blocks.

Internally, the engine that shares hardware has an additional output, and that output is associated to the last block that recognizes the last part of the common subexpression. On the other hand, the other engine connects its new input to the block which recognizes the first non-common symbol.

In the same example of the introduction (example 5), the engine 0 recognizes "$ht \bullet t \bullet p \bullet{:}\bullet/\bullet/\bullet w \bullet w \bullet w \bullet{.}\bullet e \bullet x \bullet a \bullet m \bullet p \bullet l \bullet e \bullet{.}\bullet c \bullet o \bullet m \bullet$" and the engine 1 recognizes "$ht \bullet t \bullet p \bullet{:}\bullet/\bullet/\bullet w \bullet e \bullet b \bullet - \bullet s \bullet i \bullet t \bullet e \bullet{.}\bullet o \bullet r \bullet g \bullet$". Therefore, the output of the basic block that recognizes the last symbol of the common subexpression (the last "•" of "$ht \bullet t \bullet p \bullet{:}\bullet/\bullet/\bullet w \bullet w \bullet w \bullet{.}\bullet$" in the engine 0) is connectedto the output of the engine. That "•" is associated with an "AND" block.

Example:

SET1 $http://www.example.com;\ http://www.web-site.org;$
SET2 $ftp;$ (5)

These connections can be seen in the figures 24 and 25. The figure 24 shows the additional output port of the engine 0, which is connected to the additional input port created for the engine 1. Respectively, in the figure 25 can be seen how this input is connected to the internal “AND” block that recognizes the rest of the non-common regular expression: “$w \bullet e \bullet b \bullet - \bullet s \bullet i \bullet t \bullet e \bullet . \bullet o \bullet r \bullet g \bullet$”.

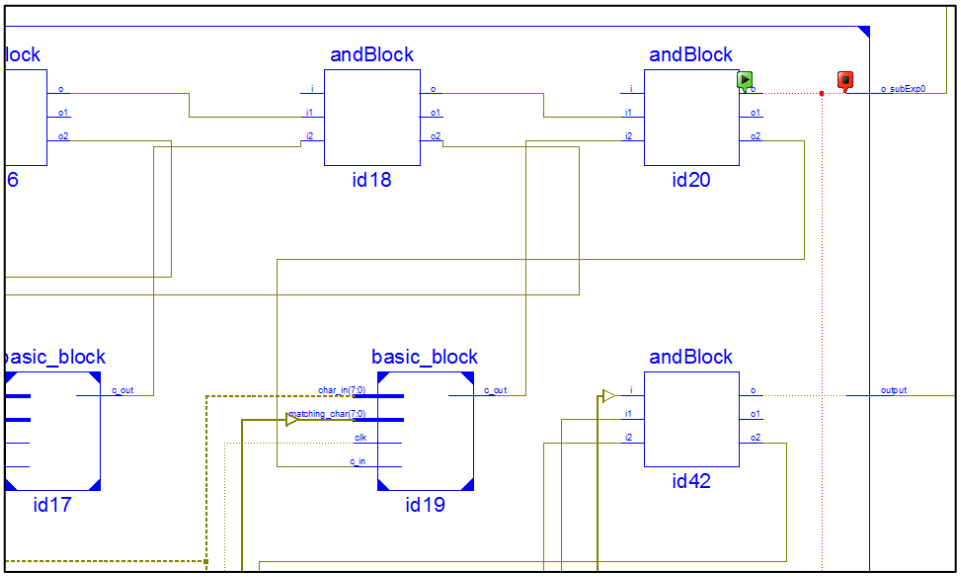

Figure 24 – Internal engine 0 connections for example 5

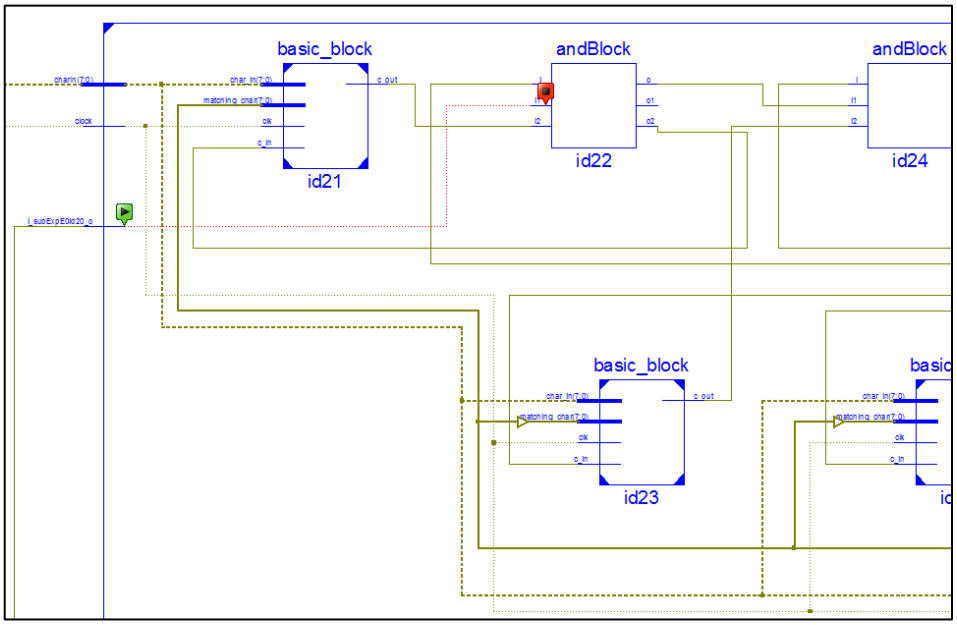

Figure 25 – Internal engine 1 connections for example 5

The algorithm performs partial engine optimization in three phases: First, every regular expression from every set is saved in its postfix form. Then, it takes one by one each regular expression and compares it with the rest, obtaining a list of possible common subexpressions for each pair, taking always the largest one from the set and saving it. And last, after comparing the regular expression with all the existing ones and obtaining a list of possible subexpressions (the largest for each pair), takes the largest one.

This process is done for all regular expressions, ensuring that every existing pair has been compared to achieve maximum overlapping.

The algorithm looks recursively for the symbols “•”, ”|” and “*”, storing every "left side" subexpression. When it finishes, the longest one of the common subexpressions list is chosen.

Below is presented the Java code used for obtaining the list of "left side" subexpressions of a regular expression (figure 26).

```
/**
 * Returns the list of subexpressions present in the regular expression given as parameter starting from the position "pos"
 */
private ArrayList<String> getSubExpressionsList(String cad, int pos) {
		ArrayList<String> subchains = new ArrayList<String>();
		String subExp1;
		String subExp2;
		String completeRE;
		ArrayList<String> listSubExp1; //Left son's subexpressions
		ArrayList<String> listSubExp2; //Right son's subexpressions
		char c = cad.charAt(pos);
		String s = new String();
		s+=c;
		switch (c){
		case '-':
				//Add the whole RE and then all subexpressions of the left son. Right son's subexpressions won't be added
				listSubExp2 = getSubExpressionsList(cad, pos-1);
				subExp2 = listSubExp2.get(0);
				listSubExp1 = getSubExpressionsList(cad, pos-subExp2.length()-1);
				subExp1 = listSubExp1.get(0);
				completeRE = new String(subExp1);
				completeRE+=listSubExp2.get(0);
				completeRE+=s;
				subchains.add(completeRE);
				subchains.addAll(listSubExp1);
		break;
		case '*':
				//Add the whole RE and then all its son's subexpressions
				listSubExp1 = getSubExpressionsList(cad, pos-1);
				subExp1 = listSubExp1.get(0);
				completeRE = new String(subExp1);
				completeRE+=s;
				subchains.add(completeRE);
				subchains.addAll(listSubExp1);
		break;
		case '|':
				// Add the whole RE and then all subexpressions, first those that belong to the left son, then those from the right son
				listSubExp2 = getSubExpressionsList(cad, pos-1);
				subExp2 = listSubExp2.get(0);
				listSubExp1 = getSubExpressionsList(cad, pos-subExp2.length()-1);
				subExp1 = listSubExp1.get(0);
				completeRE = new String(subExp1);
				completeRE+=listSubExp2.get(0);
				completeRE+=s;
				subchains.add(completeRE);
				subchains.addAll(listSubExp1);
				subchains.addAll(listSubExp2);
		break;
		default:
				// It's a char, so we add it
				subchains.add(s);
		}
		return subchains;
}
```

Figure 26 - Java code for obtaining the list of "left side" subexpressions of a regular expression

Example:

SET1 $a(aba)*;((aa)|(bb)|(cc));$

SET2 $(bba)|c;aabac;$ (6)

If we consider the four regular expression of the example 6, the algorithm will perform the following optimizations:

```
--------------------SET 0-----------------------
The expression aab·a·*· from SET 0 has been processed
No subexpressions have been reused

The expression aa·bb·|cc·| from SET 0 has been processed
The subexpression a from engine 0 associated to the block id0 has
been reused

--------------------SET 1-----------------------
The expression bb·a·c| from SET 1 has been processed
The subexpression bb· from engine 1 associated to the block id5 has
been reused

The expression aa·b·a·c· from SET 1 has been processed
The subexpression aa· from engine 1 associated to the block id2 has
been reused
```

Notice that each regular expression is showed in its postfix form (for more information about this equivalences see table in page 44):

Regular Expression	**Postfix Form**
$a(aba)*$	$aab \bullet a \bullet *$
$((aa)\|(bb)\|(cc))$	$aa \bullet bb \bullet \|cc \bullet \|$
$(bba)\|c$	$bb \bullet a \bullet c\|$
$aabac$	$aa \bullet b \bullet a \bullet c \bullet$

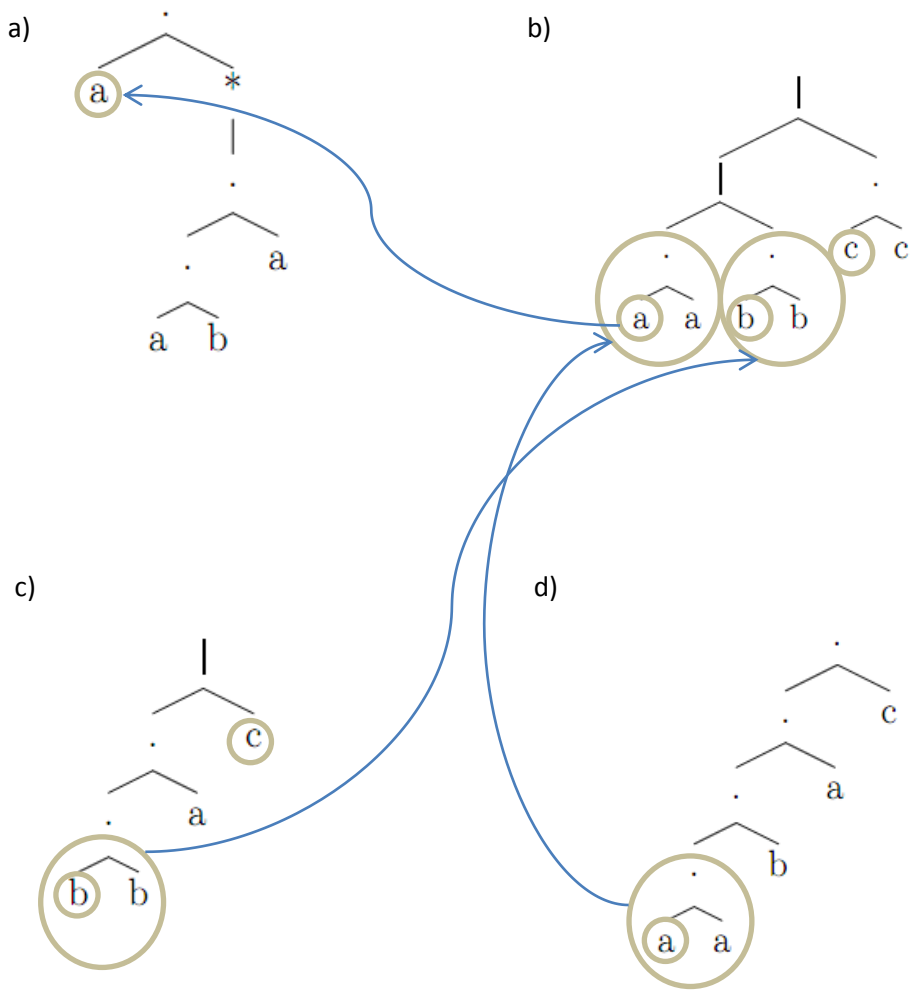

Figure 27 - Trees for a) *"$aab \bullet a \bullet *$"* , b) *" $aa \bullet bb \bullet |cc \bullet |$"*, c) *"$bb \bullet a \bullet c|$"* and d) *"$aa \bullet b \bullet a \bullet c \bullet$"*

All the possible optimizations have been rounded, and those chosen (the largest common subexpressions) have been linked.

Possible reutilizations:

- b) can use a from a).
- c) can use b, bb and c from b).
- d) can use a from a), b) and c), and aa from b).

Restrictions

It is important to remain that there are two limitations to this kind of optimization. First, right sides of regular expressions cannot be reused, and second, each block can use hardware only from one engine (although one engine can share its components to many others).

Both limitations have their origin on the sequential activation of basic blocks. Each basic block activates on next clock cycle after the basic block on its left has been activated. Therefore, we can't share hardware between engines whose associated regular expressions have a common subexpression on its right side. In that case we would not be recognizing the desired expression.

For example, let's suppose we have the expressions "ftp" and "atp", with "$ft \bullet p \bullet$" and "$at \bullet p \bullet$" as their postfix forms. In this example, the "tp" subexpression cannot be shared because the basic block that recognizes "f" must be activated before than the blocks that recognize "$t \bullet p \bullet$" to continue the recognition of "$ft \bullet p \bullet$". Thus, the basic block that recognizes "a" must be activated first or we will get false positives when an "f" is detected for "atp".

Now let's consider the regular expressions "ftk", "app" and "ftp" with "$ft \bullet k \bullet$", "$ap \bullet p \bullet$" and "$ft \bullet p \bullet$" as postfix forms. In our case, "ftp" cannot use hardware from both engines ("$ft \bullet$" from "ftk", and the last "$p \bullet$" from "app") as we cannot guarantee that "$ft \bullet$" has been recognized before "$p \bullet$", due that they are both from other independent engines and their internal blocks have different dependencies like in the example seen before.

Below are presented the experimental results obtained for various sets of regular expressions.

These benchmarks are simulating a Virtex 6 XC6VLX75T board without using the top level module described in the Code Generator chapter of the Development Environment section in page 39. Therefore, only the file “VHDLFile.vhd” is used.

In order to get accurate information about the benchmark results, a summary of the number of blocks generated has been included, along with the timing and design summary generated by the Xilinx tool in the synthesis report.

A) Lizamoon infected websites [48]:

http://lizamoon.com/ur.php
http://tadygus.com/ur.php
http://alexblane.com/ur.php
http://alisa-carter.com/ur.php
http://online-stats201.info/ur.php
http://stats-master111.info/ur.php
http://agasi-story.info/ur.php
http://general-st.info/ur.php
http://extra-service.info/ur.php
http://t6ryt56.info/ur.php
http://sol-stats.info/ur.php
http://google-stats49.info/ur.php
http://google-stats45.info/ur.php
http://google-stats50.info/ur.php
http://stats-master88.info/ur.php
http://eva-marine.info/ur.php
http://stats-master99.info/ur.php
http://world-of-books.com/ur.php

http://google-server43.info/ur.php
http://tzv-stats.info/ur.php
http://milapop.com/ur.php
http://pop-stats.info/ur.php
http://star-stats.info/ur.php
http://multi-stats.info/ur.php
http://google-stats44.info/ur.php
http://books-loader.info/ur.php
http://google-stats73.info/ur.php
http://google-stats47.info/ur.php
http://google-stats50.info/ur.php

The results obtained for this set of regular expressions are:

Block Generation Summary		
Number of engines: 29		
Number of blocks without optimizing the design: 1739		
Number of blocks generated: 1109		
Number of blocks reused: 630		
Blocks/Regular Expression: 38		
Timing Summary:		
Minimum period: 0.675ns (Maximum Frequency: 1480.385MHz)		
Minimum input arrival time before clock: 1.558ns		
Maximum output required time after clock: 2.204ns		
Maximum combinational path delay: 2.819ns		
Design Summary:		
Slice Logic Utilization:		
Number of Slice Registers:	527 out of 93,120	1%
Number used as Flip Flops:	527	

Number used as Latches:	0
Number used as Latch-thrus:	0
Number used as AND/OR logics:	0
Number of Slice LUTs:	566 out of 46,560 1%
Number used as logic:	566 out of 46,560 1%
Number using O6 output only:	552
Number using O5 output only:	0
Number using O5 and O6:	14
Number used as ROM:	0
Number used as Memory:	0 out of 16,720 0%
Number used exclusively as route-thrus:	0
Slice Logic Distribution:	
Number of occupied Slices:	168 out of 11,640 1%
Number of LUT Flip Flop pairs used:	566
Number with an unused Flip Flop:	39 out of 566 6%
Number with an unused LUT:	0 out of 566 0%
Number of fully used LUT-FF pairs:	527 out of 566 93%
Number of slice register sites lost to control set restrictions:	0 out of 93,120 0%

Ratio: 571 LUTs/29 regular expressions = 19.69 LUTs/expression

B) Phishing for the Santander bank:

@santnader.es
@bancosantnader.es
@santnader.com
@bancosantnader.es
@bnacosantander.es
@bnacosantander.com

The results obtained for this set of regular expressions are:

Block Generation Summary
Number of engines: 6
Number of blocks without optimizing the design: 194
Number of blocks generated: 103
Number of blocks reused: 91
Blocks/Regular Expression: 17
Timing Summary:
Minimum period: 0.633ns (Maximum Frequency: 1578.532MHz)
Minimum input arrival time before clock: 1.447ns
Maximum output required time after clock: 1.620ns
Maximum combinational path delay: 2.011ns
Design Summary:
Slice Logic Utilization:
Number of Slice Registers: 47 out of 93,120 1%
Number used as Flip Flops: 47
Number used as Latches: 0
Number used as Latch-thrus: 0
Number used as AND/OR logics: 0
Number of Slice LUTs: 58 out of 46,560 1%
Number used as logic: 58 out of 46,560 1%
Number using O6 output only: 56
Number using O5 output only: 0
Number using O5 and O6: 2
Number used as ROM: 0
Number used as Memory: 0 out of 16,720 0%
Number used exclusively as route-thrus: 0
Slice Logic Distribution:

Number of occupied Slices:	20 out of 11,640 1%
Number of LUT Flip Flop pairs used:	58
Number with an unused Flip Flop:	11 out of 58 18%
Number with an unused LUT:	0 out of 58 0%
Number of fully used LUT-FF pairs:	47 out of 58 81%
Number of unique control sets:	1
Number of slice register sites lost to control set restrictions:	1 out of 93,120 1%

Ratio: 58 LUTs/6 regular expressions = 9.66 LUTs/expression

C) List of known infected IPs [49]:

85.114.137.x
79.187.240.x
216.127.187.x
209.159.155.x
41.241.80.x
174.132.156.x
121.11.81.x
212.124.122.x
222.134.45.x
121.15.12.x
222.186.191.x
222.175.175.x
219.146.8.x

The results obtained for this set of regular expressions are:

Block Generation Summary
Number of engines: 13
Number of blocks without optimizing the design: 277

Number of blocks generated: 241
Number of blocks reused: 36
Blocks/Regular Expression: 18
Timing Summary:
Minimum period: 0.639ns (Maximum Frequency: 1564.700MHz)
Minimum input arrival time before clock: 1.438ns
Maximum output required time after clock: 1.620ns
Maximum combinational path delay: 2.004ns
Design Summary:
Slice Logic Utilization:
Number of Slice Registers: 110 out of 93,120 1%
Number used as Flip Flops: 110
Number used as Latches: 0
Number used as Latch-thrus: 0
Number used as AND/OR logics: 0
Number of Slice LUTs: 119 out of 46,560 1%
Number used as logic: 119 out of 46,560 1%
Number using O6 output only: 115
Number using O5 output only: 0
Number using O5 and O6: 4
Number used as ROM: 0
Number used as Memory: 0 out of 16,720 0%
Number used exclusively as route-thrus: 0
Slice Logic Distribution:
Number of occupied Slices: 40 out of 11,640 1%
Number of LUT Flip Flop pairs used: 119
Number with an unused Flip Flop: 11 out of 119 9%
Number with an unused LUT: 0 out of 119 0%
Number of fully used LUT-FF pairs: 108 out of 119 90%

Number of unique control sets:	1
Number of slice register sites lost to control set restrictions:	2 out of 93,120 1%

Ratio: 119 LUTs/13 regular expressions = 9.15 LUTs/expression

D) Set of diverse IPs

24.30.224.x
24.38.192.x
24.41.96.x
24.50.32.x
24.50.160.x
24.51.0.x
24.51.224.x
24.52.192.x
24.53.0.x
24.53.96.x
24.53.192.x
24.53.240.x
24.54.64.x
24.55.0.x
24.55.128.x
24.55.192.x
24.75.112.x
24.75.144.x
24.75.176.x
24.100.192.x
24.102.64.x
24.104.160.x
24.104.192.x
24.105.64.x
24.129.192.x
24.129.240.x

24.133.0.x

24.134.0.x

24.137.48.x

24.137.224.x

24.138.80.x

24.140.224.x

24.142.80.x

24.143.128.x

24.146.32.x

24.146.64.x

24.152.0.x

24.156.160.x

24.156.192.x

24.157.16.x

24.157.32.x

24.157.64.x

24.157.128.x

24.204.144.x

24.204.160.x

24.204.192.x

24.212.128.x

24.213.112.x

24.225.128.x

The results obtained for this set of regular expressions are:

Block Generation Summary
Number of engines: 49
Number of blocks without optimizing the design: 939
Number of blocks generated: 477
Number of blocks reused: 462
Blocks/Regular Expression: 9

Timing Summary:
Minimum period: 0.652ns (Maximum Frequency: 1532.802MHz)
Minimum input arrival time before clock: 1.503ns
Maximum output required time after clock: 2.391ns
Maximum combinational path delay: 2.580ns
Design Summary:
Slice Logic Utilization:
Number of Slice Registers: 190 out of 93,120 1%
Number used as Flip Flops: 190
Number used as Latches: 0
Number used as Latch-thrus: 0
Number used as AND/OR logics: 0
Number of Slice LUTs: 227 out of 46,560 1%
Number used as logic: 227 out of 46,560 1%
Number using O6 output only: 223
Number using O5 output only: 0
Number using O5 and O6: 4
Number used as ROM: 0
Number used as Memory: 0 out of 16,720 0%
Number used exclusively as route-thrus: 0
Slice Logic Distribution:
Number of occupied Slices: 72 out of 11,640 1%
Number of LUT Flip Flop pairs used: 227
Number with an unused Flip Flop: 37 out of 227 16%
Number with an unused LUT: 0 out of 227 0%
Number of fully used LUT-FF pairs: 190 out of 227 83%
Number of unique control sets: 1
Number of slice register sites lost to control set restrictions: 2 out of 93,120 1%

Ratio: 227 LUTs/49 regular expressions = 4.63 LUTs/expression

E) Set of 2000 randomly generated IP addresses

To obtain an experimental idea of the amount of used LUTs and the clock speed of the FPGA when pushed to its occupation limit, a small program to generate random IP addresses has been created. Therefore, we have tested it with 2000 IP addresses, obtaining the following results (note that these IPs cannot be compared to the IPs of example D, due that the IPs of this test have all digits):

Block Generation Summary
Number of blocks without optimizing the design: 51222
Number of blocks generated: 37200
Number of blocks reused: 14022
Blocks/Regular Expression: 18
Number of blocks without optimizing the design: 51222
Timing Summary:
Minimum period: 0.675ns (Maximum Frequency: 1480.385MHz)
Minimum input arrival time before clock: 1.798ns
Maximum output required time after clock: 4.334ns
Maximum combinational path delay: 4.823ns
Design Summary:
Slice Logic Utilization:
Number of Slice Registers: 16,610 out of 93,120 17%
Number used as Flip Flops: 16,610
Number used as Latches: 0
Number used as Latch-thrus: 0
Number used as AND/OR logics: 0

Number of Slice LUTs:	17,365 out of 46,560 37%
Number used as logic:	17,365 out of 46,560 37%
Number using O6 output only:	17,313
Number using O5 output only:	0
Number using O5 and O6:	52
Number used as ROM:	0
Number used as Memory:	0 out of 16,720 0%
Number used exclusively as route-thrus:	0
Slice Logic Distribution:	
Number of occupied Slices:	4,422 out of 11,640 37%
Number of LUT Flip Flop pairs used:	17,365
Number with an unused Flip Flop:	755 out of 17,365 4%
Number with an unused LUT:	0 out of 17,365 0%
Number of fully used LUT-FF pairs:	16,610 out of 17,365 95%
Number of slice register sites lost to control set restrictions:	0 out of 93,120 0%

Ratio: 17365 LUTs/2000 regular expressions = 8.68 LUTs/expression

CONCLUSIONS

Nowadays, the increasing speed of networks and storage systems make of pattern recognition a tedious task. The increasing data flow and number of patterns to be checked need of faster and flexible recognition systems. Pattern recognition is involved at many levels in the network systems. Patterns such as credit card numbers, email addresses, ports numbers, or IP addresses are clear examples of patterns that may be recognized at different levels of the network architecture.

Some systems require the analysis of many patterns for each data packet. For example: virus scanners, spam filters, traffic monitoring, email monitoring systems, hacking attacks, etc. In these examples the set of patterns to recognize increases very fast and is modified frequently.

The actual machines cannot handle this level of processing, due to the huge amount of data flow originated from high speed networks and high productivity storage systems.

In this project, a fast regular expressions recognizer has been presented, which is ideal for this daunting task. The use of a hardware implementation allows pattern recognition to be done by an independent system, decreasing the workload from systems that do this task with software programs. Moreover, the use of FPGAs for this implementation allows changing the patterns to recognize with ease without the need of buying new hardware each time we wish to do it.

Future work

The use of a code generator written in Java makes adding functionality or fixing bugs an easy task. This way we can add modules, change the hardware implementation or add a graphic interface. Moreover, a patch-based system could be added to remotely manage the application.

REFERENCES

1. http://www.xilinx.com
2. http://zone.ni.com/devzone/cda/tut/p/id/6983
3. http://www.tutorial-reports.com/computer-science/fpga
4. http://www.tutorial-reports.com/book/print/260
5. http://en.wikipedia.org/wiki/Field-programmable_gate_array
6. http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html
7. http://www.mountaingoatsoftware.com/topics/scrum
8. http://en.wikipedia.org/wiki/Scrum_(development)
9. "Introduction to languages and the theory of computation". John C. Martin. Mc Graw Hill, 1991.
10. http://www.regular-expressions.info/tools.html
11. http://en.wikipedia.org/wiki/Regular_expression
12. Badii, A. Adetoye, D. Patel, K. Hameed "*Efficient FPGA-Based Regular Expresión Pattern Matching*" (EMCIS2008)
13. B.C. Brodie, R.K. Cytron, D.E. Taylor "*A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching*" (ISCA'06)
14. Ignacio Martín Santamaría "*Evaluación de Expresiones Regulares sobre Hardware Reconfigurable*"
15. R. Sidhu, V.K. Prasanna "*Fast Regular Expression Matching Using FPGAs*" (FCCM'01)
16. J. Divashree, H. Rajashekar , Kuruvilla Varghese "*Dynamically Reconfigurable Regular Expression Matching Architecture*", 2008 International Conference on Application-Specific Systems, Architectures and Processors
17. B.C. Brodie, R.K. Cytron, D.E. Taylor "*A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching*" (ISCA'06)
18. http://en.wikipedia.org/wiki/Scrum_(development)
19. http://www.mountaingoatsoftware.com/topics/scrum

20. http://www.regexbuddy.com/
21. http://www.regexmagic.com/
22. http://en.wikipedia.org/wiki/Grep
23. http://www.powergrep.com/
24. http://delphi.com/
25. http://www.gnu.org/software/gnulib/
26. http://groovy.codehaus.org/
27. http://www.java.com/es/
28. http://en.wikipedia.org/wiki/JavaScript
29. http://www.microsoft.com/net/
30. http://www.pcre.org/
31. http://www.perl.org/
32. http://en.wikipedia.org/wiki/POSIX
33. http://en.wikipedia.org/wiki/Windows_PowerShell
34. http://www.python.org/
35. http://en.wikipedia.org/wiki/R_(programming_language)
36. http://en.wikipedia.org/wiki/Realbasic
37. http://www.ruby-lang.org/es/
38. http://en.wikipedia.org/wiki/Tcl
39. http://msdn.microsoft.com/en-us/library/t0aew7h6(v=vs.85).aspx
40. http://en.wikipedia.org/wiki/Visual_Basic
41. http://www.wxwidgets.org/
42. http://en.wikipedia.org/wiki/XML_schema
43. http://www.xquery.com/
44. http://msdn.microsoft.com/en-us/library/ms256086.aspx
45. http://www.mysql.com/
46. http://www.oracle.com/index.html

47. http://www.postgresql.org/

48. http://community.websense.com/blogs/securitylabs/archive/2011/03/31/update-on-lizamoon-mass-injection.aspx

49. http://isc.sans.org/block.txt

50. http://www.eclipse.org/

51. http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm